UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA - INE

Paulo Rogério de Pinho Filho

CONSENSO PARA ORDEM TOTAL DE TAREFAS COM PRIORIDADE

Florianópolis(SC)

2016

Paulo Rogério de Pinho Filho

CONSENSO PARA ORDEM TOTAL DE TAREFAS COM PRIORIDADE

Dissertação submetida ao Programa de Pós Graduação em Ciência da Computação para a obtenção do Grau de Mestre em Ciência da Computação.
Orientador: Luciana de Oliveira Rech, Dr.
Coorientador: Lau Cheuk Lung, Dr.

Florianópolis(SC)

2016

Catalogação na fonte elaborada pela biblioteca da
Universidade Federal de Santa Catarina

A ficha catalográfica é confeccionada pela Biblioteca Central.

Tamanho: 7cm x 12 cm

Fonte: Times New Roman 9,5

Maiores informações em:
http://www.bu.ufsc.br/design/Catalogacao.html

AGRADECIMENTOS

Agradeço ao professor Lau Cheuk Lung e à professora Luciana de Oliveira Rech pela orientação e por não ter desistido de mim. Aos professores Miguel Correia e Lasaro Camargos. Ao Hylson Vescovi Netto e aos outros colegas do Lapesd. E à Capes que financiou a pesquisa.

RESUMO

Em sistemas de tempo real, inversão de prioridades é o problema em que há tarefas de alta prioridade críticas que devem ser executadas antes de tarefas de baixa prioridade, mas não conseguem. Em sistemas distribuídos tolerantes a faltas com replicação ativa, cada réplica deve, além de resolver o problema da inversão de prioridade, executar todas as tarefas na mesma ordem para manter a consistência. Os trabalhos existentes que resolvem o problema dependem de premissas fortes como o uso de serviços de consenso, o que faz com que o algoritmo execute com uma complexidade de mensagens trocadas quadrática ou cúbica. O trabalho proposto pretende modificar algoritmos de consenso para resolver o problema de inversão de prioridades sem depender de tais premissas fortes, com complexidade de mensagens linear.

Palavras-chave: Tolerância a faltas, sistemas distribuídos, inversão de prioridade, consenso, máquinas de estado replicadas.

ABSTRACT

In real-time systems, priority inversion is the problem in which there are higher priority tasks that must be executed before lower priority taks, but cannot. In fault tolerant distributed systems with active replication, each replica must, besides solving the problem of priority inversion, execute all requests at the same order to maintain consistency. Current works that solve the problem depend on strong assumptions like the use of consensus services, that causes the algorithm to execute with quadratic or cubic complexity of message transmission. The proposed work aims to adapt consensus algorithms in order to solve the priority inversion problem without depending on such strong assumptions, achieving linear message tranmission complexity.

Keywords: Fault-tolerance, distributed systems, priority-inversion, consensus, state machine replication

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

SUMÁRIO

1 INTRODUÇÃO

1.1 MOTIVAÇÃO

Sistemas de computação frequentemente estão sujeitos a faltas, e devido a isso, sistemas confiáveis precisam lidar com tais faltas, evitando que elas se tornem erros, e mais tarde possam levar a falhas. Mascaramento de faltas é uma abordagem de tolerância de faltas. A abordagem consiste em adicionar processos redundantes que possam assumir o trabalho um do outro quando uma falta acontece.

Replicação de máquina de estado (SMR, *State Machine Replication*) 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 deve 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 cheguem no mesmo estado final e cheguem aos mesmos valores de retorno. Mesmo que algumas réplicas se desviem desse comportamento, as suas faltas são mascaradas, i.e., 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*, em cujo cerne mora o *problema do consenso*, isto é, o problema de um grupo de processos concordando em um único valor dentre um conjunto de valores propostos. Ordem total pode ser alcançada trivialmente encadeando múltiplas instâncias de consenso, cada uma para decidir a ordem de uma requisição [Hadzilacos e Toueg 1993]. Já foi demonstrado que não há algoritmo determinístico que resolva o problema de consenso sob a premissa de comunicação assíncrona [Fischer, Lynch e Paterson 1985]. Para contornar esse obstáculo algoritmos geralmente resolvem o problema de consenso garantindo que alguma propriedade de segurança apenas se mantenha se for feita alguma extensão à premissa de sincronia. Uma extensão comum é assumir sincronia mais fraca, como pela existência de um ponto no tempo a partir do qual falhas podem ser detectadas com confiança [Chandra e Toueg 1996]. Muitos algoritmos para resolver esse problema foram propostos em ambos os modelos de falhas de *crash* [Lamport 2001, Lamport 1998, Oki e Liskov 1988, Ongaro e Ousterhout 2014] e bizantinas [Castro e Liskov 1999, Kotla et al. 2007].

Talvez o algoritmo de consenso mais bem conhecido existente seja o protocolo do Sínodo [Lamport 1998]. Este algoritmo resolve o problema de consenso garantindo que se a maioria dos processos aceitar um valor proposto, então nenhum valor diferente pode ser

aceito no seu lugar. Esse protocolo é usado como bloco de construção para o algoritmo de ordenação total de requisições, Paxos, em que cada processo tem uma sequência de valores, ou requisições, e cada valor da sequência é decidido por meio de uma execução do protocolo do Sínodo. Paxos garante segurança enquanto tolera f faltas em um sistema com $2f + 1$ réplicas [Lamport 1998]. Apesar de ser muito conhecido e utilizado, Paxos também é famoso por ser de difícil compreensão devido a sutilezas do seu funcionamento, e não prover um fundamento detalhado para a sua implementação [Ongaro e Ousterhout 2014]. Para suprir essa necessidade foi desenvolvido o Raft [Ongaro e Ousterhout 2014], de melhor compreensão e pronto para implementação, enquanto mantém a capacidade de tolerar f faltas com $2f + 1$ processos em ambiente assíncrono do Paxos.

Sistemas confiáveis podem ter requerimentos de tempo real, i.e., podem precisar que operações sejam executadas antes de prazos rígidos [Verissimo e Rodrigues 2001]. Nesses sistemas geralmente as tarefas são organizadas em um cronograma para que sejam executadas antes do seu *deadline*, o tempo limite permitido para a sua execução. Atender tal problema, contudo, 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, os processadores se comunicam por meio de canais que não garantem a entrega de mensagens dentro de um prazo, ou mesmo dentro da ordem, tornando a comunicação inconfiável e comprometendo a previsibilidade do sistema. Nesses casos, a alternativa é um escalonamento *soft*, para atingir requerimentos menos estritos de tempo real [Locke 1986]. Nesse tipo de sistema de tempo real, é comum atribuir *prioridades* a requisições. Tarefas de prioridade mais alta são mais críticas e devem ser executadas antes das tarefas de prioridade baixa.

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, Yeo e Venugopal 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 e Liskov 1999, Kotla et al. 2007], serviços de *backup* cooperativo [Aiyer et al. 2005], e sistemas de gerência de bancos de dados relacionais [Luiz, Lung e Correia 2011].

O problema de difusão de ordem total de requisições com prioridades em um sistema de máquinas de estado é chamado de PB-SMR (*Priority-Based State Machine Replication*). Há alguns trabalhos que lidam com esse problema [Nakamura e Takizawa 1992, Rodrigues, Veríssimo e Casimiro 1995, Wang et al. 2002]. O assunto apareceu primeiro em [Nakamura e 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 falhas. O algoritmo de [Rodrigues, Veríssimo e Casimiro 1995] usa *view atomic multicast* [Schiper e 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 considerável número de mensagens trocadas entre as réplicas, e como resultado, têm complexidade de mensagens de $O(n^3)$ e $O(n^2)$ respectivamente.

1.2 OBJETIVO

O objetivo desse trabalho é resolver o problema PB-SMR (*Priority-Based State Machine Replication*), replicação de máquina de estado baseado em prioridade, sem qualquer premissa mais forte do que *eventual synchrony* [Dwork, Lynch e Stockmeyer 1988], isto é, sem considerar que há um ponto desconhecido de antemão no tempo em que o sistema se comporta como se fosse síncrono. Evita-se a delegação a serviços de consenso que prodigalizam o número de mensagens, adaptando-se algoritmos de consenso bem conhecidos: Paxos [Lamport 1998] e Raft [Ongaro e Ousterhout 2014]. A adaptação foi feita diretamente dos algoritmos originais para que os novos algoritmos tratem o problema de prioridade sem precisar de tais serviços de consenso. O resultado são algoritmos que resolvem o problema possuindo complexidade de número de mensagens linear, sendo assim mais eficiente que os algoritmos relacionados.

Com vista a alcançar o objetivo geral, foram definidos os seguintes objetivos específicos:

1. Avaliação do estado da arte na área de consenso com prioridade;
2. Proposição de um algoritmo que adapte o Paxos para lidar com prioridades;
3. Proposição e desenvolvimento de um algoritmo que adapte o Raft para tratar prioridades;
4. Análise comparativa dos algoritmos propostos com os trabalhos relacionados;
5. Implementação e realização de uma avaliação experimental de um dos algoritmos propostos, comparando-o com o algoritmo original do qual ele foi derivado.

1.3 CONTRIBUIÇÕES

As principais contribuições deste trabalho são as seguintes:

1. Um algoritmo de replicação de máquinas de estado baseado em prioridade para resolver o problema de consenso levando em conta requisições com prioridades: PRaxos,

um algoritmo baseado no Paxos.

2. O algoritmo de replicação de máquinas de estado baseado em prioridade para resolver o problema de consenso levando em conta requisições com prioridades: PRaft, um algoritmo baseado no Raft.

3. A implementação do PRaft e uma avaliação comparativa com o Raft.

1.4 ORGANIZAÇÃO DO TEXTO

Neste capítulo estão a contextualização do trabalho, as motivações para a sua realização, o problema, os objetivos e as contribuições. O restante do trabalhos está organizado da seguinte maneira:

- *Capítulo 2: Conceitos Básicos:* Em que se fala dos conceitos necessários para sustentar o trabalho proposto: computação distribuída, serviços, tolerância a faltas, tempo real e prioridades.

- *Capítulo 3: Trabalhos Relacionados:* Onde é apresentada a revisão dos trabalhos relacionados de consenso com prioridades , comparando-os e focando no número de passos que realizam para executar uma requisição.

- *Capítulo 4: Proposta:* Em que são apresentados e descritos os dois algoritmos propostos desenvolvidos nesse trabalho: o PRaft e o PRaxos, e uma seção informal de provas para a correção do PRaxos.

- *Capítulo 5: Testes:* Onde estão os resultados dos testes e as avaliações experimentais incluindo comparações quantitativas entre o PRaft e o Raft.

- *Capítulo 6: Conclusões:* Avaliação dos resultados obtidos com a realização desse trabalho em relação aos objetivos e às contribuições e sugestões de trabalhos futuros sobre o tema de sistemas de máquina de estados baseados em prioridades.

2 CONCEITOS

Para facilitar a compreensão desta dissertação, neste capítulo serão expostos os conceitos relacionados à proposta dessa dissertação. São abordados neste capítulo os seguintes conceitos: máquina de estados replicada, consenso, Paxos, Raft, BFT (*Bizantine Fault Tolerance*) e inversão de prioridades. Dois trabalhos bem conhecidos para a solução de consenso são o Paxos [Lamport 1998,Lamport 2001] e o PBFT (*Practical BFT*) [Castro e Liskov 2002]. O Paxos foi desenvolvido para ambiente de faltas de crash, e o PBFT para faltas bizantinas.

2.1 REPLICAÇÃO DE MÁQUINA DE ESTADO

Replicação de Máquina de Estado (*State Machine Replication*) [Schneider 1990] é uma técnica para implementar serviços tolerantes a faltas. Serviços baseados em um único processador sujeito a sofrer faltas corre o risco de não atender ao seu contrato. Uma maneira de fazer o serviço tolerante a faltas é por meio da *replicação*. O serviço passa a ser um sistema distribuído com diversos servidores. A Replicação de Máquinas de Estado é um tipo de replicação ativa, isto é, em que todos os servidores executam juntos o mesmo serviço. Assim o sistema tolera a falha de alguns servidores enquanto os servidores que permanecem corretos garantem o funcionamento do serviço.

Um serviço executa *operações* requisitadas por *clientes*. Operações podem ter *parâmetros* e, quando executadas, podem gerar valores de *retorno* que são enviados para os clientes requisitantes. Cada processador tem um *estado*. À medida em que executa operações, o processador pode transitar de um estado para o outro.

Para que a replicação de máquina de estado funcione, o sistema deve ter três características: em primeiro lugar, todos os processadores devem começar com o mesmo estado inicial. Esse estado depende do serviço prestado e varia de aplicação para aplicação. Em segundo lugar, todas as operações oferecidas pelo serviço devem ser *determinísticas*, isto é, operações que, executadas diversas vezes com os mesmos parâmetros e sobre o mesmo estado, sempre retornam o mesmo valor e sempre transitam o sistema para o mesmo estado. E em terceiro lugar, todos os processadores corretos executam as operações na mesma ordem.

Um processador que executa uma operação determinística com o mesmo estado e com os mesmos parâmetros gerará sempre o mesmo valor de retorno e transitará sempre para o mesmo estado.

Obedecendo esses critério, os processadores são *réplicas* uns dos outros, mantendo

sempre o mesmo estado, executando sempre a mesma sequência de operações.

Essas características devem ser levadas em consideração ao implementar um serviço de replicação de máquinas de estado, e podem ser facilmente implementadas. O único problema que não é trivial de resolver é a ordem de execução, que é um problema de consenso.

2.2 PROBLEMA DE CONSENSO

O problema de consenso [Dwork, Lynch e Stockmeyer 1988] é aquele em que os processadores de um sistema distribuído devem entrar em acordo acerca de um valor ao mesmo tempo em que o sistema tolera faltas. Se o sistema é uma máquina de estado replicada, então o valor do acordo é a posição em que uma requisição deve tomar na sequência de requisições. Já que todas as réplicas devem executar a mesma sequência de requisições. A falha em chegar ao acordo faria com que a sequência de cada processo fique diferente, e aí não haveria mais replicação, impossibilitando o sistema de tolerar faltas.

O sistema é formado por um conjunto N de processadores $p_1, p_2, ..., p_n$. Cada processador p_i tem um valor v_i no domínio V. Além disso, cada processador pode estar *correto* ou *falho*. Um processador correto é aquele que opera normalmente, enquanto o processador falho pode se comportar de maneira não prevista pelo algoritmo.

O problema de consenso se define dessa maneira:

1. Todos os processadores corretos devem *decidir* o mesmo valor.
2. Se o valor inicial de todos os processadores é o mesmo, então esse é o valor que deve ser decidido.

Algoritmos que se propõem a resolver o problema de consenso devem levar em consideração ainda outros dois fatores: o modelo de faltas e a resiliência.

2.2.1 Modelo de Falta

Em primeiro lugar, deve ser decidido qual é o modelo de faltas. Quais são as faltas que podem ocorrer a um processador? Há três tipos de faltas.

1. *Crash*. O processador pode falhar por falta de *crash* se ele suspender o seu funcionamento permanentemente ou temporariamente a qualquer hora.
2. *Omission*: O processador pode falhar por falta de *omission* se as suas mensagens deixarem de ser transmitidas temporariamente a qualquer hora.
3. *Byzantime*: O processador pode falhar por falta *Byzantine* se ele passa a se comportar de maneira arbitrária em execução ou em mensagens transmitidas.

2.2.2 Modelo de Tempo

Em segundo lugar, deve ser decidido o modelo de tempo. Basicamente isso significa achar o valor de duas constantes, Δ e Φ, e decidir se esses valores são conhecidos de antemão. O valor Δ significa o tempo limite para a transmissão de mensagens, e Φ significa a razão de *drift* (o tempo por segundo que um relógio corre mais rápido que o outro) máxima entre os relógios dos processadores.

Se há valores definidos para Δ e Φ e esses valores são conhecidos de antemão, então o modelo de tempo é *síncrono*. Se esses valores são desconhecidos de antemão, ou se eles não são definidos, então o modelo de tempo é *assíncrono*.

2.2.3 Resiliência

Para cada um dos casos em que se associa o problema de consenso a um modelo de falta e a um modelo de tempo deve-se encontrar um algoritmo que resolva o problema com uma certa *resiliência*, ou uma prova de que o algoritmo não existe. A resiliência do algoritmo é uma função entre o número total de processadores n e o número total de faltas que ele tolera f.

Um algoritmo que resolve o problema de consenso no modelo de faltas de *crash* em um modelo de tempo *assíncrono* tem a resiliência representada da equação

$$n = 2f + 1,$$

que quer dizer que o sistema tolera que uma minoria de seus processadores faltem para funcionar corretamente.

Um algoritmo no modelo de faltas bizantino tem resiliência

$$n = 3f + 1.$$

2.2.4 Propriedades

Propriedades são os objetivos que o algoritmo alcança e estão relacionados à definição do problema de consenso. As propriedades são divididas em *Segurança* e *Terminação*. Segurança quer dizer que nada de ruim jamais vai acontecer, e terminação quer dizer que algo de bom sempre vai acontecer [Alpern e Schneider 1985]. As propriedades são separadas assim porque não existe algoritmo que resolva consenso em modelo de tempo *assíncrono*. [Fischer, Lynch e Paterson 1985] Portanto, os algoritmos têm as suas provas garantindo as propriedades de segurança em modelo de tempo assíncrono e

garantindo as propriedades de terminação em um modelo de tempo mais relaxado.

2.3 PAXOS

Paxos [Lamport 1998] [Lamport 2001] é um algoritmo que resolve o problema de consenso baseado num modelo de faltas de *crash* e no modelo de tempo *assíncrono*.

Os processadores são classificados como *Proposer*, *Acceptor* e *Learner* [Lamport 2001]. Um processador *proposer propõe* um valor para processadores *acceptors*. Um valor é *escolhido* quando um número suficiente grande de *acceptors aceitou* o valor, então processadores *learners aprendem* que valor é esse. O algoritmo é esquematizado na Figura 1.

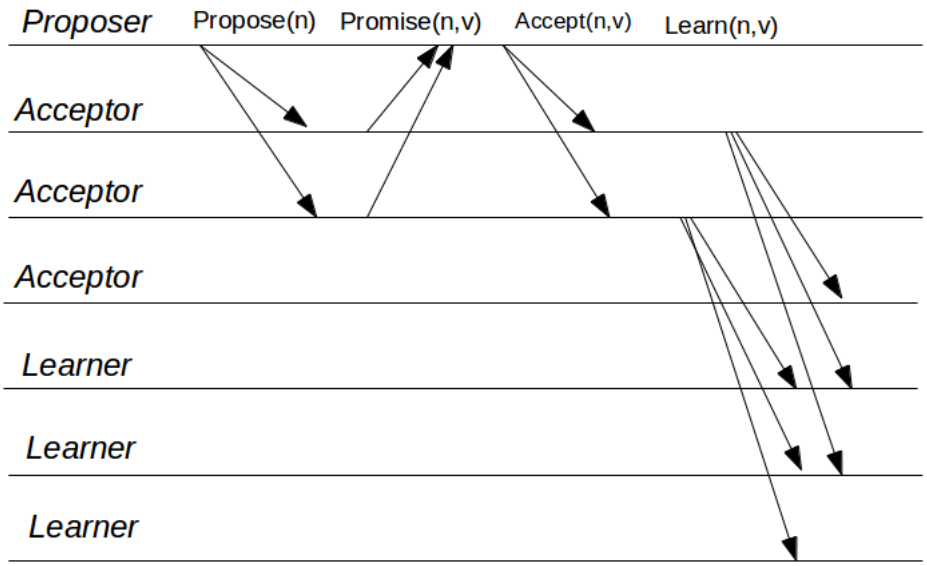

Figura 1: Algoritmo Paxos

As propriedades de *segurança* são:

- 1. Apenas valores *escolhidos* são *aprendidos*.
- 2. Apenas um valor *proposto* é *escolhido*.

A propriedade de *terminação* é:

- 3. Todos os *learners* corretos (que não sofrem falta) *aprenderão* o valor *escolhido*.

As classes podem ser representadas por quaisquer processadores no sistema, possivelmente todos os processadores desempenham função de todas as classes.

No Paxos, um valor *aceito* é o equivalente a um valor *decidido* no problema de consenso. O problema de consenso estará resolvido quando a propriedade de terminação for satisfeita.

O valor inicial dos processadores não importa para a solução do problema, pois o que será aceito é um valor proposto por um processador diferenciado. A qualquer hora um processador pode fazer uma proposta do seu valor para o sistema. Esse processador é classificado como *proposer*. Cada valor proposto é associado a um número de proposta s que o diferencia dos demais.

Para que o valor de apenas uma proposta seja escolhido, a maioria dos *acceptors* deve aceitá-la. Essa maioria constitui um *quorum*. Antes, contudo, de fazer a proposta do seu valor, o *proposer* deve saber se algum valor já foi escolhido.

O *proposer* deve escolher um *quorum* Q e enviar uma mensagem $Propose(n)$ para cada *acceptor* em Q. Um *acceptor* aceita a mensagem apenas se o número de proposta dela for maior que o seu próprio n. Ao aceitar, envia uma mensagem $Promise(n, v)$, onde v é o valor, dentre os que ele já aceitou, com o maior número de proposta.

Ao receber as respostas do *quorum*, o *acceptor* escolhe, dentre os valores enviados, o valor v que tem o maior número de proposta e envia uma mensagem $Accept(n, v)$ ao *quorum*. Assim o *proposer* não muda o valor que já foi escolhido anteriormente. Se nenhum valor foi escolhido até então, então o *proposer* não encontrará entre as mensagens *Promise* recebidas nenhum valor e estará livre para transmitir o *Accept* com o seu próprio valor inicial.

Um *acceptor*, ao receber $Accept(n, v)$, se ele já não recebeu um *Propose* com um número de proposta maior que n, *aceita* v e envia uma mensagem $Learn(n, v)$ para os *learners*. Um *learner*, ao receber essa mensagem de um *quorum*, *aprende* v.

2.4 PAXOS EM MÁQUINA DE ESTADO REPLICADA

O Paxos pode ser utilizado em sistemas de máquinas de estado replicadas mediante algumas adaptações que não mudam o sentido do algoritmo. Neste caso um processador é eleito como líder, sendo o líder o único processador que pode assumir o papel de *proposer*. Todos os processadores assumem os papéis de *acceptor* e de *learner*.

Como um processo está sujeito a falhas, para que um serviço se mantenha no ar, é necessário que haja redundância de processos. Para atender a esta necessidade, sistemas são desenvolvidos de acordo com o modelo de replicação de máquinas de estados. Nesse modelo os processos iniciam no mesmo estado e todas as operações que realizam alteram os seus estados de forma determinística. Desse modo, se todos os processos executam o mesmo conjunto de tarefas, todos ficam no mesmo estado, e estão prontos a substituir o processo principal.

Para manter os estados consistentes, os processos precisam de um protocolo que os façam designar na sua lista de execução a cada tarefa a mesma posição.

As propriedades do algoritmo são:

1. Acordo: não há dois processos que designam tarefas diferentes para a mesma posição na lista.

2. Validação: uma vez que um dos processos tenha designado uma tarefa em alguma posição, todos os processos devem designá-la na mesma posição.

3. Terminação: todos os processos acabam por designar todas as tarefas requisitadas às suas posições.

O protocolo deve garantir que essas propriedades sejam atendidas para resolver o problema de consenso. Além disso, ele deve levar em consideração que o sistema está configurado de acordo com um modelo de tempo e um modelo de falta.

O modelo de tempo em questão é o assíncrono. Os processos se comunicam por troca de mensagens, as quais não podem ser alteradas, mas podem ser duplicadas, se perder ou tomar um tempo arbitrário para chegar ao seu destino. Além disso o sistema deve entrar em intervalos em que se comporta com sincronia, em que o envio das mensagens é garantido e dentro de um limite de tempo.

O modelo de faltas é *crash-recovery*. Os processos podem falhar por *crash*, e podem então voltar ao funcionamento.

O Paxos atende às três propriedades do problema de consenso dentro dos modelos descritos.

Há n processos no sistema, um líder e um conjunto de $n-1$ réplicas. Paxos é baseado em quorum, portanto uma tarefa está confirmada quando uma maioria dos processos define a sua posição na lista. O sistema funciona corretamente se até f processos estiverem falhos ao mesmo tempo, sendo que $n = 2f + 1$.

O líder é eleito por meio de um protocolo de eleição de líder. Então ele tem que estabelecer a sua visão para que as réplicas o reconheçam como líder. À medida que o líder recebe tarefas dos clientes, ele designa a cada tarefa uma posição na lista de execução e envia uma mensagem às réplicas para cada tarefa e a sua respectiva posição. Quando uma réplica recebe a mensagem do líder, responde imediatamente, e uma vez que o líder tem a resposta da maioria dos processos, está confirmada a posição da tarefa na lista de execução, e o líder envia uma mensagem de confirmação às réplicas. Uma sequência de tarefas com ordem já confirmada pode ser executada pelo líder e a resposta de cada execução transmitida ao cliente requisitante.

O algoritmo segue os passos como ilustrado na figura 2.

Passo 1. O líder é eleito por meio de um protocolo de eleição de líder e deve estabelecer a sua visão. Ele escolhe um número de visão v e faz um *broadcast* da mensagem $prepare(v)$ para as réplicas.

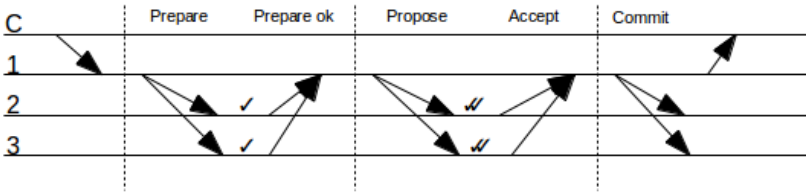

Figura 2: Paxos em máquinas de estado replicadas.

Passo 2. Ao receber $prepare(v)$, o processo p, se estiver operando sob uma visão $u < v$, aceita a nova visão e envia uma mensagem $prepare\_ok(v, V_p)$ ao líder. V_p é uma lista de votações que p fez até então. Cada votação vai com a forma (r_i, v_i), em que r_i é a tarefa votada na posição i da lista, e v_i é a visão em que r_i foi votado.

Se a visão de p for maior ou igual a v, então p não aceita a visão. O processo p envia uma resposta com a sua visão para que o processo requisitante possa refazer a proposta de visão com uma visão maior.

Ao receber uma maioria de respostas $prepare_ok$, o líder estabelece a visão. Agora ele tem a lista de votação de um quorum de processos. Se em qualquer lista houver o registro da votação de uma requisição r_i na posição i, então todos os processos que ainda não fizeram este registro devem fazê-lo. Em caso de conflito, se na lista V_p houver o registro (r_i, v_i) e na lista V_q o registro (r'_i, v'_i), então a votação confirmada é aquela em que consta a maior visão. O líder executa o passo 3 para cada posição com votos registrados para garantir que a maioria dos processos tenha votado.

Passo 3. Tendo recebido um quorum de aceites, o líder reformula a fila de pronto P_l, incluídas as tarefas novas. Então o líder faz *broadcast* de $proposal(v, P_l)$.

Passo 4. Recebida a proposta, o processo p responde $accept(v, r, p)$ ao líder.

Passo 5. Tendo o líder recebido um quorum de aceites, ele faz *broadcast* de $success(v, r)$, executa a tarefa da requisição r e envia a resposta ao cliente requisitante.

Passo 6. Ao receber a confirmação, o processo p insere (r_n, v_n) na sua lista de votações V_p.

2.5 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 complexidade. Compreendê-lo é um desafio para muitas pessoas, mesmo pesquisadores experientes [Ongaro e 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 3 ilustra a sequência de passos do algoritmo desde a eleição até a confirmação de requisições. 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.

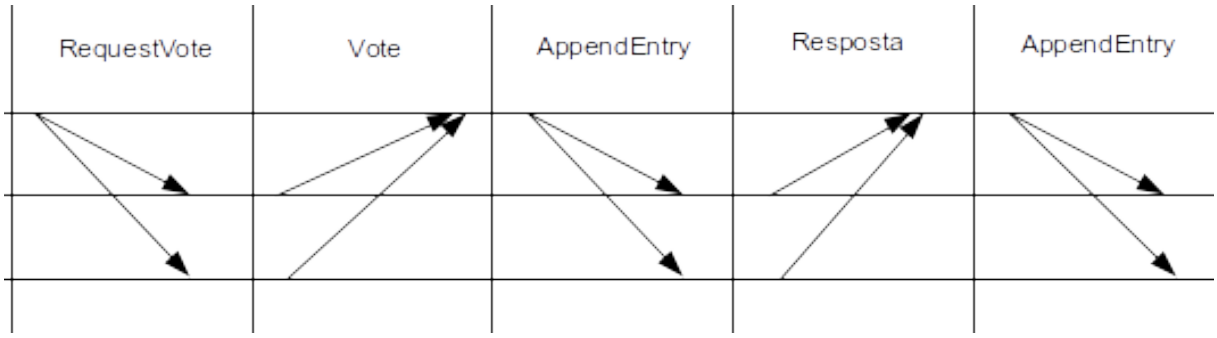

Figura 3: Sequência de passos do Raft

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.

Cada processo mantêm uma lista de entradas chamada *log*, onde tem todas as requisições que já recebeu. Ao timeout, 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, 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 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.

Cada entrada do *log* de um processo tem a 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 a resposta 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 o *log* dos dois são iguais até aquele ponto. Mas 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 $next\_index$ diminui, maior é a sublista de requisições enviadas ao seguidor no *AppendEntry* para replicação, e isso se repete até chegar o ponto em que o *log* dos dois são iguais e a replicação é bem sucedida.

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.

2.6 BFT

BFT (*Byzantine Fault Tolerance*) é um protocolo de consenso para replicação de máquinas de estado tolerante a faltas bizantinas [Castro e Liskov 2002], [Lamport, Shostak e Pease 1982].

Nem todas as faltas que sofre um serviço são de *crash*. Pode haver invasões, bugs, ou outros problemas imprevistos que alteram o comportamento de um processo. Ao comportamento arbitrário decorrente desse tipo de falha dá-se o nome de falta bizantina.

Apesar do problema em questão ser o mesmo problema de consenso resolvido pelo Paxos, e compartilhar do mesmo modelo de tempo, o algoritmo tem modificações significativas devido ao modelo de faltas bizantinas. No Paxos os processos confiam nas mensagens que recebem, mas no modelo de faltas bizantinas isso não é possível.

O problema é representado abstratamente pelo exército bizantino cercando uma cidade inimiga. O exército é dividido em vários pelotões comandados por generais, um dos quais pode ser um traidor. Os generais, para ter sucesso no ataque, precisam se coordenar para que todos os generais leais ataquem ao mesmo tempo. Se acontecer de uma parte atacar e outra parte recuar, eles perdem. A coordenação acontece por meio de troca de mensagens.

Suponha que o exército é dividido entre um comandante e dois tenentes. O comandante envia uma mensagem aos seus tenentes para atacar. Cada tenente, suspeitando que o comandante é um traidor, troca mensagens entre si para saber qual ordem o outro recebeu. Esse problema não tem solução, porque se um tenente que recebeu uma ordem de ataque viu que o outro tenente recebeu uma ordem para recuar, ele não tem como saber se foi o comandante que enviou ordens contrárias a cada um dos tenentes, ou se foi o outro tenente que lhe mostrou uma falsificação da ordem do seu comandante.

Para resolver esse problema, é necessário mais um tenente envolvido. Suponha que o comandante é o traidor e enviou duas ordens de ataque e uma ordem de recuar. Os tenentes, suspeitando do comandante, trocam mensagens entre si para mostrar qual ordem cada um recebeu. O tenente que recebeu a ordem de recuar, vendo que os outros tenentes receberam ordens de atacar, tem certeza que o traidor é o general, e decide que ele vai atacar junto com os outros. Os tenentes que receberam ordem de atacar não precisam saber se o traidor é o general ou o tenente que lhes mostrou ordem de recuar, pois ele têm certeza que um dos dois é leal e vai participar do ataque.

A decisão é tomada de acordo com a ordem mostrada pela maioria. Se a proporção entre traidores e generais leais é de um para quatro, os traidores são uma minoria que não pode sabotar os generais leais mostrando ordens contraditórias na segunda fase.

O algoritmo PBFT, que está ilustrado na Figura 4, foi desenvolvido para resolver o problema dos generais bizantinos.

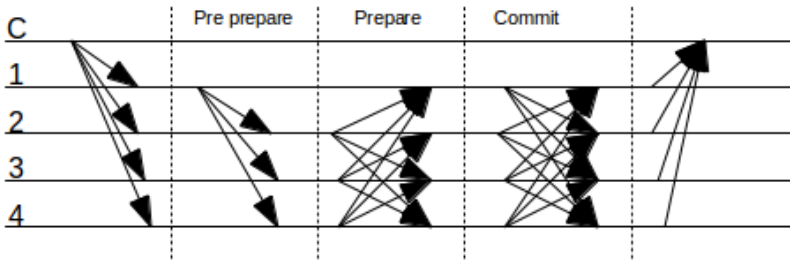

Figura 4: BFT.

No PBFT os passos descritos são *Prepepare* e *Prepare*. Em adição a esses, há mais um passo *Commit* que está aí para garantir a consistência através de mudanças de visão.

2.7 INVERSÃO DE PRIORIDADES

A abordagem deste trabalho utiliza o conceito de inversão de prioridades de [Wang et al. 2002]. Cada tarefa enviada por clientes é associada a uma prioridade, de modo que $Priority(\alpha) < Priority(\beta)$ se a prioridade de α é menor do que a prioridade de β.

Um processo que executa a tarefa α enquanto β é recebida pelo sistema está em inversão de prioridade local. Quando um número suficiente de processos estiver em inversão de prioridade local, então o sistema está em inversão de prioridade em grupo. O número exato depende do problema.

Um sistema de replicação garante que um número máximo v de respostas é enviado para o cliente. O cliente junta as respostas e escuta uma função de validação para obter o resultado definitivo da requisição. Em um sistema em modelo de faltas de *crash*, $v = 1$, pois basta a resposta de um processo ao cliente. Em um sistema em modelo de faltas bizantinas, o cliente não pode confiar apenas na resposta de um processo, mas precisa receber a mesma resposta correta da maioria, portanto, $v = f + 1$.

Um sistema está em inversão de prioridade em grupo quando, dentre todos os processos que responderam a uma requisição, não há v processos que não estejam em inversão de prioridade local. Isso significa que, para que o sistema não entre em inversão de prioridade em grupo, pelo menos v réplicas sem inversão de prioridade local deve responder ao cliente para cada requisição.

2.8 CONSIDERAÇÕES FINAIS

Neste capítulo foram apresentados os conceito necessários para compreender a proposta dessa dissertação. A compreensão dos conceitos de replicação de máquinas de estado e consenso são essenciais pois neles será implantada a prioridade. Paxos e Raft são algoritmos bem difundidos na solução de consenso para máquinas de estado replicadas, mas não implementam prioridades. O conceito de BFT foi apresentado devido às possibilidades de implementação de prioridades no contexto de faltas bizantinas, e essas possibilidades são comentadas no final da dissertação.

No próximo capítulo serão apresentados os trabalhos relacionados que abordam consenso com prioridades para máquinas de estado replicadas.

3 TRABALHOS RELACIONADOS

Este capítulo apresenta os principais trabalhos relacionados a consenso com uso de prioridade. O pioneiro é o algoritmo denominado PriTO [Nakamura e Takizawa 1992], seguido por um algoritmo que utiliza sincronia virtual [Rodrigues, Veríssimo e Casimiro 1995]. Segue um trabalho [Wang et al. 2002] que define o conceito de inversão de prioridade em grupo e por fim um trabalho [Miedes e Munoz-Escoı 2007] que apresenta de maneira mais simples o tratamento da inversão de prioridade.

3.1 PRITO

O algoritmo PriTO [Nakamura e Takizawa 1992] (priority-based total ordering) é uma difusão de ordem total no qual mensagens são entregues a membros de um grupo, segundo as suas prioridades. Existem três estruturas nas quais as mensagens são manipuladas: RRL, PRL e ARL, que significam *accepted*, *pre-acknowledge* e *acknowledge* (a partir de agora, referenciados como: aceita, pré-reconhecida e reconhecida). Uma mensagem comum segue um fluxo por estas três estruturas, na sequência: RRL, PRL, e ARL: uma requisição é aceita (entra na RRL), é pré-reconhecida se foi aceita por cada membro do grupo, e é reconhecida se foi pré-reconhecida por cada membro do grupo.

A priorização de mensagens ocorre da seguinte maneira: durante a passagem de mensagens de RRL para PRL, a mensagem prioritária é inserida na devida ordem, de acordo com a prioridade. A passagem de PRL para ARL é feita retirando-se o elemento de PRL que está no início da fila. Um exemplo na Tabela 1 ilustra essa operação. Inicialmente (linha 1), os pedidos a, b e c foram recebidos em RRL. Estes pedidos foram inseridos em PRL, estrutura que mantém-se ordenada de acordo com as prioridades. O próximo pedido a ser inserido é o pedido d_2. Na linha 2, d_2 é inserido na devida ordem. Ainda, suponha que o pedido a foi pré-reconhecido pelo grupo; daí então, insere-se a_0 em PRL. Na linha 3, o pedido e_2 é inserido na devida ordem; b foi pré-reconhecido pelo grupo. Os pedidos e_2 e d_2 possuem a mesma prioridade, mas d_2 foi inserido primeiro, por isso e_2 encontra-se após d_2. Na linha 4, f_2 é inserido na devida posição, e c foi pré-reconhecido pelo grupo. Na linha 5, d foi reconhecido pelo grupo; a foi pré-reconhecido quando d foi aceito, logo a é reconhecido Porém, a permanece em PRL porque existem requisições de maior prioridade na fila. Por fim, na linha 5, g_1 é inserido na devida posição, depois de a, pois a foi reconhecido. Na linha 6, o pedido h_4 foi inserido na devida posição. Na linha 7, o pedido i_2 foi inserido na devida posição. Como c foi pré-reconhecido quando f foi aceito, então c é reconhecido Sendo c a requisição que está no topo de PRL, o pedido c_4

Tabela 1: Exemplo de ordenação com prioridade no PRiTO [Nakamura e Takizawa 1992]

linha	ARL	PRL	RRL	Próximo
1		$c_4\ b_3\ a_1$	$a_0\ b_0\ c_0$	d_2
2		$c_4\ b_3\ \boldsymbol{d_2}\ a_1\ a_0$	$b_0\ c_0\ d_0$	e_2
3		$c_4\ b_3\ d_2\ \boldsymbol{e_2}\ a_1\ a_0\ b_0$	$c_0\ d_0\ e_0$	f_2
4		$c_4\ b_3\ d_2\ e_2\ \boldsymbol{f_2}\ a_1\ a_0\ b_0\ c_0$	$d_0\ e_0\ f_0$	g_1
5		$c_4\ b_3\ d_2\ e_2\ f_2\ a_1\ \boldsymbol{g_1}\ a_0\ b_0\ c_0\ d_0$	$e_0\ f_0\ g_0$	h_4
6		$c_4\ \boldsymbol{h_4}\ b_3\ d_2\ e_2\ f_2\ a_1\ g_1\ a_0\ b_0\ c_0\ d_0\ e_0$	$f_0\ g_0\ h_0$	i_2
7	c_4	$h_4\ b_3\ d_2\ e_2\ f_2\ \boldsymbol{i_2}\ a_1\ g_1\ a_0\ b_0\ d_0\ e_0\ f_0$	$g_0\ h_0\ i_0$	j_3

é transferido da *PRL* para a *ARL*.

Pedidos de baixa prioridade, mesmo que estejam reconhecidos, podem não ser entregues à aplicação, caso requisições de alta prioridade estejam ocorrendo com frequência. Assim, requisições de baixa prioridade pode sofrer de inanição. Em um trabalho estendido [Nakamura e Takizawa 1993], esse problema é resolvido por meio de um particionamento no tempo, formando uma *janela de execução*. Requisições de uma *janela de execução* não podem ser deslocadas para outra *janela de execução*.

A tolerância a faltas deste trabalho é muito rudimentar: caso uma entidade do grupo falhe, o grupo é encerrado.

3.2 CONSENSO COM PRIORIDADE E SINCRONIA VIRTUAL

Rodrigues [Rodrigues, Veríssimo e Casimiro 1995] apresenta um protocolo de consenso para sistemas de replicação de máquinas de estado. O protocolo suporta o envio de mensagens prioritárias, e tolera falta por meio do uso de um serviço de comunicação de grupo confiável, obtendo assim uma sincronia virtual. O serviço de comunicação de grupo será responsável pela transmissão de mensagens, gerenciamento de visão, tolerância a faltas de paradas dos processos e difusão atômica.

O protocolo funciona como descrito a seguir. Cada processo mantém uma lista de requisições ordenadas. A prioridade de uma requisição pode ser alta ou baixa. No topo da lista estão as de alta prioridade, e ao fundo estão as mensagens de baixa prioridade. Os processadores executam as requisições a partir do topo da lista. Quando o sistema recebe uma requisição de baixa prioridade, ela é simplesmente colocada no fundo da lista de todas as réplicas. Quando recebe uma requisição de alta prioridade, ela é posta no topo da lista da réplica mais rápida. Nas outra réplicas, ela é posta na posição correspondente.

A Figura 5 mostra um exemplo onde ocorre inserção de mensagem prioritária. Existem três processos $p_{1..3}$ que possuem as filas de requisições conforme mostra a Figura 5(a). O processador $p1$ é o mais rápido do conjunto; neste processador a mensagem m, de baixa prioridade, já foi executada. Quando a mensagem h, de alta prioridade, chega ao sistema,

ela será inserida no topo da lista do processador mais rápido: o processador $p1$. Nos processadores $p2$ e $p3$, a mensagem h será inserida nas posições correspondentes à posição em que h foi inserida em $p1$. O estado das listas após a inserção de h em todos os processos está ilustrado na Figura 5(b).

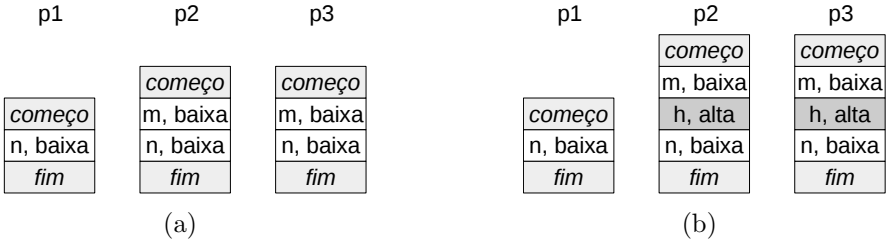

Figura 5: Inserção de uma mensagem prioritária: (a) antes e (b) depois [Rodrigues, Veríssimo e Casimiro 1995].

O algoritmo não permite preempção, isto é, que requisições em execução sejam interrompidas para a execução de uma requisição mais prioritária. As réplicas mais lentas, antes de executar a nova requisição de alta prioridade, precisam executar todas as requisições de mais baixa prioridade que a réplica mais rápida já havia executado até então. Por isso, o algoritmo não interrompe requisições de baixa prioridade para executar um requisição de alta prioridade. As requisições são confirmadas quando a maioria dos processos termina a sua execução, e só então a resposta é enviada ao cliente. Devido a isso e à falta da capacidade de interrupção, requisições de alta prioridade precisam esperar a execução e a confirmação das requisições de baixa prioridade que já foram executadas pela réplica mais rápida. O algoritmo trata inversão de prioridade, mas não permite *rollback*, pois basta que uma réplica processe uma requisição para que a sua posição esteja definida e todas as outras réplicas a executem na mesma posição. Apenas as requisições que não foram executadas por nenhuma réplica podem mudar de posição.

Como se pode observar, este algoritmo, ao contrário do PriTO, é tolerante a faltas de *crash*, mas ainda não permite preempção. Essas duas características são buscadas por este trabalho para que as requisições de alta prioridade sejam executadas logo que cheguem ao sistema.

3.3 INVERSÃO DE PRIORIDADE EM GRUPO

O trabalho de [Wang et al. 2002] propõe uma formalização mais rigorosa do problema de inversão de prioridade para sistemas distribuídos, definindo GPI, *Group Priority Inversion*, a partir da definição familiar de LPI, *Local Priority Inversion*. Em poucas palavras, LPI acontece quando uma única réplica sofre inversão de prioridade; e GPI acontece quando a maioria das réplicas está em LPI. O algoritmo desenvolvido resolve o problema de consenso evitando GPI em todos os casos, o que significa que, do ponto de

vista do cliente, requisições nunca causarão inversão de prioridade, e o algoritmo é capaz de fazer preempção. O algoritmo depende de um serviço de consenso

O funcionamento do protocolo é descrito desta forma: Cada vez que o sistema recebe uma requisição r, um serviço de consenso é invocado para definir o número de sequência da nova requisição, isto é, qual posição na sequência de requisições ela ocupa. Requisições que são de prioridade mais baixa que r são movidas do seu lugar para a posição seguinte, para dar espaço para que r ocupe posição correta na sequência, respeitando a ordem de prioridade. Se uma réplica já havia começado a executar quaisquer requisições entre as que haviam sido movidas, então a réplica interrompe a sua execução e restaura o estado da sua máquina de estados para 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, o que quer dizer que outra requisição de prioridade mais alta não pode tirá-lo do seu lugar. Então a resposta de r é enviada ao cliente requisitante.

O algoritmo é modular, de tal forma que qualquer protocolo de consenso pode ser utilizado, e o tratamento das prioridades é feito em outro módulo. A Figura 6 mostra a arquitetura da proposta. O módulo de ordem total (TO) é responsável por receber as requisições enviadas pelos clientes e garantir a tolerância a faltas de parada. O módulo de agendamento prevê inversões de prioridade permitindo a mudança na ordem total inicialmente definida pelo módulo TO.

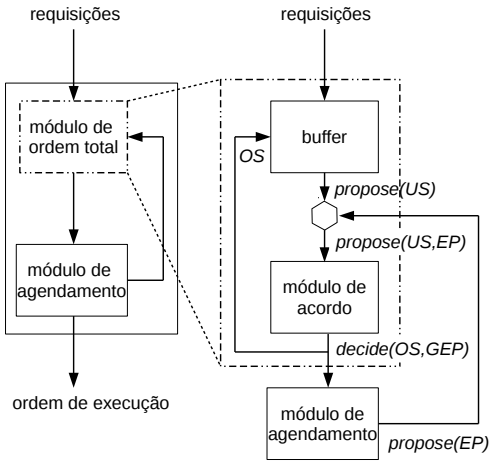

Figura 6: Arquitetura para implementar consenso com prioridade [Wang et al. 2002].

No detalhe do módulo TO, um *buffer* gerencia as requisições, e um módulo de acordo é responsável por ordenar os pedidos. O *buffer* contém requisições sem ordem, representado pela sigla US (*Unordered Set*). Existe um controlador de progresso da execução, denominado EP, que juntamente com o US compõe a entrada para o módulo de acordo. A saída do módulo de acordo é um conjunto de pedidos ordenados, denominado OS (*Ordered Set*). O comando *propose* solicita ao módulo de acordo a ordenação de pedidos, cuja saída é obtida por meio da primitiva *decide*.

Cada réplica mantém uma sequência de requisições e dois marcadores. Um marcador LEP que indica o progresso local de execução, e um marcador GEP que indica o progresso de execução do grupo. GEP é definido como o menor LEP dentre as $f + 1$ réplicas com maior LEP. Todas as requisições em posições menores ou igual ao GEP estão confirmadas. Portanto as requisições confirmadas são aquelas que foram executadas pela maioria das réplicas. As requisições que ficam antes do GEP estão definidas e não mudam mais de posição. As que estão depois do GEP estão sujeitas a mudar de posição se uma requisição de prioridade mais alta for recebida de um cliente. Quando isso acontece, a réplica invoca um consenso para definir o GEP e a sequência com a nova requisição. Se uma réplica tinha executado requisições que sofreram inversão de prioridade, então ele faz *rollback* dessas requisições e reassume a execução a partir do seu LEP re-estabelecido. Se uma dessas requisições que sofreram inversão de prioridade era a que ele estava executando no momento, ele interrompe a execução e faz *rollback*. Quando uma réplica termina uma execução, ela invoca o consenso para definir o GEP. Assim, para um sistema processe uma requisição, são necessárias pelo menos $f + 1$ invocações de consenso.

Este algoritmo é o mais parecido com a proposta deste trabalho, pois ele não só resolve o problema de consenso com prioridades, como ainda permite preempção. Além disso a sua formalização de inversão de prioridade em grupo é útil para a definição e solução do problema, que será usado neste trabalho. A diferença entre o algoritmo e este trabalho é que este algoritmo, assim como o [Rodrigues, Veríssimo e Casimiro 1995], depende de um serviço de consenso que faça esse trabalho para ele, o que aumenta a complexidade de mensagens consideravelmente, como se verá mais adiante.

3.4 ACRESCENTAR PRIORIDADE A PROTOCOLOS EXISTENTES

Miedes [Miedes e Munoz-Escoı 2007] apresenta diversos algoritmos para implementar prioridade em um serviço de difusão atômica com ordem total. Dentre os algoritmos, existe um para adicionar prioridades a algoritmos que usam sequenciador fixo. A ideia apresentada é descomplicada: manter uma lista de pedidos que chegam, e antes de enviá-los às réplicas, anexar uma marcação de prioridade aos mesmos. A lista é ordenada pela prioridade. Assim, o sequenciador envia sempre o primeiro item da fila às réplicas. Esse mecanismo contém em si o problema da inanição de pedidos de baixa prioridade. Uma solução para este problema é o uso de um temporizador, que após a finalização da contagem de tempo, bloqueia o recebimento de novos itens, envia todos os pedidos que estão na lista, e em seguida desbloqueia o sistema, recebendo a partir de então novos pedidos.

3.5 CONSIDERAÇÕES FINAIS

Neste capítulo foram apresentados os principais trabalhos relacionados ao problema.

PriTo [Nakamura e Takizawa 1992] depende 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.

[Rodrigues, Veríssimo e Casimiro 1995] assume que toda a comunicação entre os processos é feita por um sistema subjacente de comunicação de grupo, que será responsável pela tolerância a faltas do sistema. Os trabalhos propostos nessa dissertação (PRaxos e PRaft) evitam a inversão de prioridade interrompendo e reordenando requisições não confirmadas.

A principal diferença entre [Wang et al. 2002] e o presente trabalho é a sua principal motivação de não usar um algoritmo genérico de acordo, o que faz as réplicas trocarem mais mensagens do que no PRaft e o PRaxos. Em [Wang et al. 2002], dada uma lista de requisições para executar, o algoritmo de consenso é invocado uma vez, mais pelo menos $f + 1$ vezes para cada requisição que confirma. Ou seja, para cada requisição confirmada, o algoritmo invoca o serviço de consenso $f + 2$ vezes, sendo que em cada chamada do serviço de consenso, os processsos trocam pelo menos $3(f + 1)$ mensagens. No 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.

A solução que Miedes [Miedes e Munoz-Escoı 2007] apresenta é simples e útil para compreensão básica de um processo de priorização, mas é uma solução que não tolera faltas.

A proposta apresentada nessa dissertação se diferencia dos demais trabalhos mencionados por não depender de premissas, mas por estender algoritmos de ordem total já consagrados, como o Paxos e o Raft, adicionando aos mesmos a capacidade de tratar mensagens prioritárias. O problema de inversão de prioridade também é tratado em ambas as implementações.

A Tabela 2 mostra a comparação entre os algoritmos estudados. O algoritmo de [Rodrigues, Veríssimo e Casimiro 1995] usa *view atomic multicast* [Schiper e 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 e Takizawa 1992] tem complexidade apenas quadrática, mas foi feito para sistemas síncronos. Os algoritmos desenvolvidos, tendo sido feito à semelhança dos algoritmos

Tabela 2: Comparação com os algoritmos relacionados

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

originais, Raft [Ongaro e Ousterhout 2014] e Paxos [Lamport 2001], executam em tempo linear. No próximo capítulo será apresentada a proposta, com a descrição dos algoritmos desenvolvidos. Maiores detalhes sobre a comparação entre os algoritmos desenvolvidos e os trabalhos relacionados encontram-se na seção 4.4.

4 PROPOSTA

Neste capítulo será tratada a proposta deste trabalho. O objetivo consiste em desenvolver dois algoritmos para resolver o problema de consenso para máquinas de estado replicadas baseadas em prioridades, adaptando dois algoritmos conhecidos para esse problema: o Paxos [Lamport 2001] e o Raft [Ongaro e Ousterhout 2014].

Primeiro se apresentará o modelo do problema, com os nomes das variáveis, definições e as propriedades que definem o problema. Então será apresentada a descrição do algoritmo PRaxos, seguido da sua prova. Depois será apresentada a descrição do algoritmo PRaft.

4.1 MODELO DE SISTEMA

O sistema é composto por um conjunto de Π processos, dos quais $f < |\Pi|/2$ podem falhar. 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', $P(r) > P(r')$.

Os seguintes predicados expressam estados de uma requisição. Se um processo p aceitou uma requisição r, então $Accepted(r, p)$ é verdadeiro, e $r \in \texttt{learned}_p$.

Assume-se um modelo de falha em que os processos que falharam não se recuperam.

A comunicação entre as réplicas é feita 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, *Global Stabilization Time*), chamado *eventual synchrony* [Cristian e Fetzer 1998, Fetzer e 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 tempo suficiente 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 preparada para execução e p está executando 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 objetivo dos algoritmos é difundir requisições aos processos que os executam e então responder ao cliente requisitante.

Serão agora apresentadas as propriedades que definem o problema de consenso para máquinas de estado replicadas baseadas em prioridades para o PRaxos e para o PRaft. Os dois algoritmos não compartilham da mesma notação, portanto essas propriedades serão descritas usando as notações próprias para cada um.

Para o PRaxos, adere-se à nomenclatura do algoritmo Paxos, que é a sua base. Um processo faz a *difusão* de uma mensagem m invocando $Propose(m)$ e um processo p *entrega* uma mensagem adicionando-na no final da sequência $\texttt{learned}_p$, inicialmente vazio. O elemento aprendido na posição i denota-se $\texttt{learned}_p[i]$. O algoritmo PRaxos satisfaz as seguintes propriedades de segurança:

- Não-Trivialidade: Se uma requisição $r \in \texttt{learned}_p$ para qualquer processo $p \in \Pi$, então r foi proposto.

- Acordo: Se quaisquer dois processos p e q em Π aprenderam requisições no número de sequência i, então $\texttt{learned}_p[i] = \texttt{learned}_q[i]$.

- Ordem de Prioridade: Requisições prontas são escolhidas segundo sua prioridade.

Cada uma dessas propriedades será atendida pelo algoritmo como se verá pelas provas que o seguem.

A seguir estão as propriedades de segurança que o algoritmo PRaft satisfaz:

- 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.

4.2 PRAXOS

Nesta seção é apresentada a descrição do PRaxos. PRaxos é um algoritmo de consenso baseado no Paxos. Como algoritmo de consenso, o Paxos tem por objetivo resolver o problema de ordem total de máquinas de estado replicadas. Em adição a isso, PRaxos foi desenvolvido para tratar requisições com prioridades, processando requisições de alta prioridade antes das requisições de baixa prioridade.

Adicionar prioridades às máquinas de estado replicadas cria a necessidade de adaptar o protocolo a novas restrições. Na medida em que os pedidos de alta prioridade chegam ao sistema, pedidos de baixa prioridade recebidos anteriormente precisam ser postos atrás, na sequência de execução, do pedido recém-chegado.

No PRaxos, um pedido somente é aceito após ter sido executado pela maioria dos processos, enquanto outros pedidos estão *prontos*, e sua posição pode mudar na medida em que pedidos de alta prioridade chegam. *Acceptors* em algum momento aceitarão os valores escolhidos, sendo que o líder é sempre o primeiro a aceitar.

Pedidos são ordenados de acordo com sua prioridade, e nesta ordem são executados pelos processos. Quando processos estão executando um pedido *pronto*, e recebem um pedido de alta prioridade r, então o sistema entra em uma *inversão de prioridade*. Processos interrompem todos as requisições *prontas* de baixa prioridade que estão executando e retornam o sistema ao estado anterior à sua execução. Então, a sequência de pedidos é reordenada, colocando r antes dos outros pedidos, e a operação então é retomada.

Como exemplo, suponha que um processo tem a sequência $\{\alpha_1, \beta_4, \gamma_2, \delta_2\}$. O pedido α é selecionado, a execução de β's e γ's termina, e δ encontra-se em execução. As prioridades destes processos estão subescritas. Então, chega um novo pedido ϵ_3. O processo interrompe δ e retorna o estado do sistema ao ponto anterior à execução de γ. Depois da reordenação, a sequência resultante é $\{\alpha_1, \beta_4, \epsilon_3, \gamma_2, \delta_2\}$.

A Figura 7 mostra o comportamento do protocolo quando os papéis são distribuídos aos processadores. O processo p_1 é o líder e atua como um *proposer* distinto, os demais processos são *acceptor*s, e além desses papéis, todos agem simultaneamente como *learner*s. O algoritmo PRaxos realiza três passos. Processadores executam processos depois de receber a mensagem *Accept* e o valor de retorno é enviado para o cliente solicitante por p_1, depois que o pedido for aprendido. A aprendizagem pode ser atingida pelo recebimento de mensagens *Learn*, por uma maioria de *learner*s, ou pela verificação da sequência de pedidos que cada *acceptor* enviou com as mensagens *Promise*.

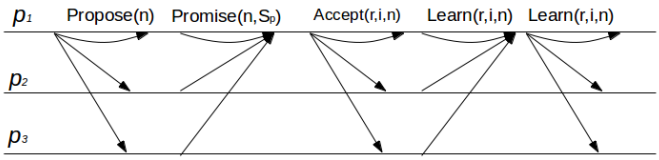

Figura 7: PRaxos na visão de processadores.

4.2.1 Algoritmo

Todos os processos no sistema, incluindo o líder, atuam como *acceptor*s e *learner*s, portanto a sequência $p.S$ em ambos será a mesma. Para facilitar a explicação a seguir, estas regras serão tratadas separadamente.

4.2.1.1 O Líder

O Algoritmo 1 descreve como o líder funciona. O líder inicia enviando $\langle Propose, n\rangle$ com um número de proposta n para todos os *acceptor*s. Um processo *acceptor* p que recebe esta mensagem responde se, e somente se, o número da última proposta por ele recebida for menor que o número da proposta que a mensagem contém, ou seja, se $p.n < n$. O *acceptor* configura $p.n = n$ e responde para o líder, enviando $\langle Promise, p.S, n\rangle$, juntamente com sua sequência de pedidos.

O líder espera pela resposta de todos os processos (linhas 3-6), ou até um timeout expirar. Se o líder conseguiu reunir a respostas suficientes para formar um quorum de $f + 1$ processos (linha 7), então o próximo passo inicia.

Neste ponto, o líder possui um conjunto Q que consiste na sequência $p.S$ de cada *acceptor* no quórum. Isto provê ao líder informação sobre os pedidos que cada réplica recebeu, aceitou e aprendeu. O próximo passo é determinar quais pedidos foram escolhidos e quais estão prontos, o que será descrito a seguir.

Para cada número de sequência i (linha 8), um pedido r é escolhido em i se existem $f + 1$ sequências S em Q tal que $S[i] = r$, ou um dos processos tenha aprendido r em i (linha 9). Este caso é denotado como $Chosen(i)$. Se nenhum pedido foi escolhido em i, então um dos casos a seguir se aplica: existem $f + 1$ sequências S em Q tais que $S[i] = Null$ ou existem $f + 2$ sequências S em Q tais que $S[i]$ é atribuído a diferentes pedidos (linha 12). De qualquer maneira, este caso é denotado como $Free(i)$. Apesar da aparente oposição entre as duas expressões, $\neg Chosen(i)$ não é o mesmo que $Free(i)$, visto que há um caso no qual o número de sequência i é simultaneamente $\neg Chosen(i)$ and $\neg Free(i)$.

O líder realiza as seguintes ações, de acordo com o número de sequência:

Algoritmo 1 Proposta do líder

1: $n \leftarrow n + 1$
2: Envia $\langle Propose, n\rangle$ aos *acceptor*s
3: **repeat**
4: Recebe $\langle Accept, p.S, n\rangle$ do *acceptor* p
5: $Q \leftarrow Q \cup \{p.S\}$
6: **until** $|Q| = N$ **or** *time out*
7: **if** $|Q| \geq f + 1$ **then**
8: **for** $i = 0$ **do**
9: **if** $(\exists R \subseteq Q, |R| \geq f + 1 \wedge \exists r \forall S \in R, S[i] = r) \vee (\exists p.S \in R, Learned(p.S[i], p))$ **then**
10: Aprende requisição aprendida em (Q, i)
11: Envia $\langle Learn, n, r, i\rangle$ aos outros *learner*s
12: **else if** $\exists R \subseteq Q, (|R| \geq f + 2 \wedge \neg(\exists S, S' \in R, S[i] = S'[i])) \vee (|R| \geq f + 1 \wedge \forall S \in R, S[i] = Null)$ **then**
13: $\mu \leftarrow i$
14: interrompe for-loop
15: **else**
16: $r \leftarrow S[i] \in Q$ onde $\neg(\exists S'[i] \in Q, S'[i].n > S[i].n)$
17: Envia $\langle Accept, n, r, i\rangle$ aos *acceptor*s
18: **end if**
19: $i \leftarrow i + 1$
20: **end for**
21: Cria uma sequência *Ready* com as requisições em Q de μ em diante, e as requisições em $Buffer$.
22: Ordena *Ready* segundo as prioridades.
23: **for** $i = \mu$ **do**
24: $l.S[i] \leftarrow Ready[i]$
25: **end for**
26: **end if**

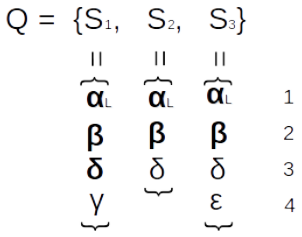

Figura 8: O líder determina pedidos *Chosen* e *Free*. Pedidos em negrito foram executados. A letra L ao lado do pedido significa que o pedido foi aprendido.

1. Se $Chosen(i)$, então o líder *aprende* o pedido escolhido (linha 10).

2. Se ambos $\neg Chosen(i)$ e $\neg Free(i)$, então o líder envia $\langle Accept, n, r, i\rangle$ para todos os *acceptor*s, tal que r é o pedido de maior número em $S[i]$, entre as sequências em Q, isto é, com o maior $S[i].n$ (linhas 16, 17).

3. Se $Free(i)$, então o líder encerra a iteração (linhas 13, 14).

Quando um líder *aprende* um pedido r na posição i, então $Learned(r, i)$ se torna verdade e o líder envia a mensagem $\langle Learn, n, r, i\rangle$ para os demais *learner*s. Um processo *learner* que recebe esta mensagem também aprende r em i.

Quando o líder encerra a iteração, uma marca $\mu = i$ é configurada na sequência (linha 13). A sequência de $S[0]$ até $S[\mu-1]$ é chamada de seção $Processada$, e a sequência a partir de $S[\mu]$ é chamada de seção $Pronta$. Pedidos na seção $Processada$ não sofrem de inversão de prioridade, visto que não há posição i em $Processadas$ tal que $Free(i)$, portanto podendo haver $Chosen(i)$. Como não há posição i em $Prontas$ tal que $Chosen(i)$, pedidos nesta seção podem causar inversão de prioridade.

O líder reúne todos os pedidos na seção $Processadas$ de cada S em Q e sequências que acompanham estes pedidos, e ordena os pedidos de acordo com as prioridades (linhas 21, 22). A seguir, o líder envia $\langle Accept, r, i, n\rangle$ a todos os *acceptor*s, para cada um dos pedidos, iniciando na posição $i = \mu$ (linhas 23-25).

A Figura 8 mostra um exemplo do procedimento. Suponha que o sistema tem um total de três processos, o que configura $f = 1$, e o líder reuniu respostas de todos os processos. Na primeira iteração, $i = 1$, todos os processos executaram e aprenderam α, portanto o número de sequência 1 é $Chosen(1)$. Da mesma maneira, todos os processos executaram β no número de sequência 2, o que define $Chosen(2)$. No número de sequência 3, $f + 1$ sequências não foram executadas com nenhum pedido, o que define $Free(3)$, e $\mu = 3$. O líder encerra a iteração neste ponto.

Para obter o número de sequência resultante, suponha que $P(\delta) = 5$, $P(\gamma) = 3$ e que o líder tenha recebido do cliente o pedido ϵ, com prioridade 4. A sequência resultante seria $S = \{\alpha, \beta, \delta, \epsilon, \gamma\}$.

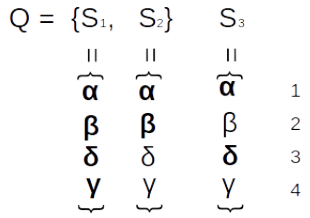

Figura 9: Exemplo de como *Chosen* e *Free* não são iguais quando nem todas as réplicas corretas respondem à proposição do líder. Pedidos em negrito foram executados.

A Figura 9 ilustra outro caso. Suponha que o sistema possui três processos, e que o total de faltas toleradas seja $f = 1$. Os processos p_1 e p_2 responderam à proposição do líder enviando mensagem de promessa, e a resposta enviada por p_3 foi perdida na transmissão. De acordo com as sequências recebidas, α, β e δ foram escolhidas, mas δ não foi considerada escolhida pelo líder. Os predicados $Chosen(1)$ e $Chosen(2)$ são verdadeiros porque α e β foram executados e aceitos por $f+1$ réplicas em Q. Entretanto, os pedidos δ e γ não foram executados nem aceitos, e portanto não existem $f+1$ aceites nulos, ou $f+2$ aceites diferentes em Q para os números de sequência 3 e 4: $Free(3)$ e $Free(4)$ são falsos. O líder envia $\langle \delta, 3, n \rangle$ e $\langle \gamma, 4, n \rangle$ para os *acceptor*s e considera estes pedidos escolhidos. O número de sequência $Free$ neste exemplo é 5, e consequentemente $\mu = 5$.

O Algoritmo 2 descreve como o líder trata as requisições dos clientes. O líder mantém armazenados os pedidos dos clientes. Se o líder recebe um pedido r de prioridade menor do que os pedidos da seção $Prontos$, o líder envia $\langle Accept, r, i, n \rangle$ para os *acceptor*s, tal que i é a primeira posição disponível no fim da sequência (linhas 6, 7).

Quando o líder recebe um pedido r que possui prioridade maior que todos os pedidos que estão na seção $Prontos$, o líder insere r no *buffer* (linhas 3, 4). Após a exaustão de um temporizador, o líder inicia uma nova proposta enviando $\langle Propose, n+1 \rangle$.

Algoritmo 2 *Loop* principal do líder

```
1:  loop
2:    Recebe requisição r do cliente c.
3:    if P(r) > P(End(l.S)) then
4:      Buffer ← Buffer ∪ {r}
5:    else
6:      Append r em l.S
7:      i ← size of l.S
8:      Envia ⟨Accept, n, r, i⟩ aos acceptors
9:    end if
10: end loop
```

4.2.1.2 A Réplica

As réplicas recebem as mensagens *Accept* e *Learn* do líder, visto que as réplicas exercem ambos os papéis de *acceptor* e *learner*.

Uma réplica *acceptor*a p executa pedidos na sua própria sequência $p.S$ na ordem em que eles aparecem. Quando p conclui a execução do pedido $r = p.S[i]$, ela aceita r e envia ao líder a mensagem $\langle Learn, p.n, r, i\rangle$. Neste ponto, o pedido faz parte do conjunto $Accepted(r, i, p)$. Então, p executa o pedido na posição $p.S[i + 1]$.

O líder armazena as mensagens *Learn* provenientes das réplicas. Quando o líder recebe $f+1$ $\langle Learn, r, i, n\rangle$ com os mesmos parâmetros, ele aprende r e envia $\langle Learn, r, i, n\rangle$ para os demais *learner*s.

Se o *acceptor* p atinge um ponto no qual $p.S[i]$ não tem pedidos, mas existe um item $p.S[i + k]$, k posições adiante, que contém um pedido, então p envia uma mensagem para o líder solicitando o pedido que está faltando no número de sequência i.

O Algoritmo 3 descreve como um *acceptor* recebe pedidos. Quando o *acceptor* p recebe uma mensagem $\langle Accept, r, i, n\rangle$ do líder, ele move todos os pedidos em $p.S[j]$ para $p.S[j + 1]$, com j iniciando em i, até o fim da sequência (linhas 3, 4), para criar um espaço e incluir r na sequência (linha 10). Todos os pedidos movidos foram executados, mas foram desfeitos e tornaram-se $\neg Accepted(r, i, p)$ (linha 8). Se p estivesse executando qualquer pedido que foi movido, p interromperia a execução (linhas 5, 6), iria desfazer as operações do pedido e iniciar a execução de $p.S[i]$ (linhas 11, 12).

Algoritmo 3 Aceite do *acceptor*

```
 1: Recebe ⟨Accept, r, i, n⟩ do líder
 2: if p.n = n then
 3:    for j = |p.S| decrementando até i do
 4:       p.S[j + 1] ← p.S[j]
 5:       if Estiver executando p.S[j] then
 6:          Interrompe p.S[j]
 7:       end if
 8:       Rollback p.S[j]
 9:    end for
10:    p.S[i] ← r
11:    if p.π >= i then
12:       p.π ← i
13:    end if
14: end if
```

Quando um *acceptor* p recebe uma mensagem $\langle Learn, n, r, i\rangle$ do líder, ele aprende r em i. Se $p.S[i] \neq r$, então p envia uma mensagem para o líder, solicitando o pedido. O *acceptor* p receberia uma mensagem *Accept* e deveria agir como previamente explicado. Por fim, p iria aprender o pedido.

4.2.1.3 Eleição de Líder

O líder é o único processo que armazena pedidos do cliente em memória. Se o líder falha, estes pedidos serão perdidos. Um processo tem um limite de tempo para comunicar-se com o líder. Se esse tempo esgota-se, o processo inicia um protocolo de eleição de líder. O novo líder então começa a receber os pedidos dos clientes e comporta-se conforme o protocolo especificado.

4.2.2 Provas

Esta seção apresenta as provas da correção do algoritmo. Prova-se que o PRaxos satisfaz as propriedades de Não-Trivialidade, Acordo e Ordem de Prioridade.

Para provar a Não-Trivialidade e o Acordo, demonstra-se dois lemas que estão relacionados às propriedades de segurança do problema de consenso.

Lemma 1 *Apenas requisições escolhidas são aprendidas.*

É demonstrado que se uma requisição foi aprendida por um *learner*, então ela foi antes escolhida por *acceptor*s. Há quatro meios de um processo *learner* aprender uma requisição r no número de sequência i:

1. quando o *learner* recebe a mensagem $\langle Learn, r, i, n\rangle$ de $f + 1$ *acceptor*s;
2. quando o *learner* recebe uma mensagem $\langle Learn, r, i, n\rangle$ do líder;
3. quando o *learner* é um líder que acabou de propor e: ou a requisição r foi aceita em i na sequência de $f + 1$ processos que responderam com mensagem *promise*,
4. ou havia uma requisição aprendida r em i na sequência de um processo que respondeu com uma mensagem *promise*.

No primeiro caso, um processo que envia a mensagem *Learn* ao *learner* já havia recebido a mensagem *Learn* do líder, que acontece quando o líder recebe *Accept* da maioria dos *acceptors*. Se o líder recebeu essa mensagem de $f+1$ *acceptors*, então $f+1$ *acceptor*s haviam executado e aceitado r, portanto r estava escolhido.

No segundo caso, o líder envia a mensagem $Learn(r, i, n)$ depois que aprendeu r, e, como demonstrado no primeiro caso, isso implica que r estava escolhido.

No terceiro caso, r foi aceito por $f + 1$ *acceptor*s, portanto estava escolhido.

No quarto caso, r havia sido aprendido pela maioria dos processos através dos três casos anteriores, e portanto estava escolhido. □

Lemma 2 *Para cada posição na sequência, apenas uma requisição proposta é escolhida.*

Isso é demonstrado mostrando que nenhuma requisição em números de sequência $Free(i)$ estão escolhidas, e que requisições em números de sequência $\neg Free(i)$ estão seguras de não ser substituídas por outras requisições.

Como em Paxos, se a maioria dos *acceptor*s aceita r, então r é *escolhido*. Antes de mandar qualquer proposta, o *proposer* tem de perguntar à maioria dos processos *acceptor*s quais são as requisições que eles haviam aceitado até então. Como pelo menos um *acceptor* que havia aceitado uma requisição escolhida está incluído na maioria, o *proposer* está seguro de receber todas as requisições escolhidas nas mensagens $Promise$, e deve propor essas requisições escolhidas nos seus respectivos números de sequência. Para propor diferentes requisições num número de sequência i, a maioria reunida pelo *proposer* deve não incluir qualquer *acceptor* que tenha aceitado uma requisição em i. Isso garante que apenas uma requisição é escolhida para cada número de sequência.

Em PRaxos, o líder envia mensagens *Accept* para cada nova requisição quando ela não sofre inversão de prioridade de qualquer requisição pronta, comportando-se exatamente como o Paxos. Ou seja, quando a requisição recebida não tem prioridade maior que a prioridade de qualquer requisição pronta na sequência. Contudo, se uma requisição que sofre inversão de prioridade é recebida, o líder inicia uma nova proposta enviando a $Propose$ aos *acceptor*s. Neste caso, para cada número de sequência i na sequência que foi $Chosen(i)$, pelo menos um *acceptor* no quorum havia aceitado a requisição escolhida, e o líder tem de propor aquela requisição no número de sequência i. Requisições prontas só são propostas no número de sequência i se $Free(i)$. Portanto, requisições em $Free(i)$ não estão escolhidas.

Há casos em que o *proposer* juntou respostas da maioria dos *acceptor*s em que menos da maioria havia aceitado r no número de sequência i, o que quer dizer que r não está escolhido em i. O *proposer* pode distinguir se r está escolhido ou não se:

1. a maioria incluía $f + 1$ *acceptor*s que não haviam aceitado nenhuma requisição,

2. ou a maioria incluía $f + 2$ *acceptor*s que haviam aceitado requisições diferentes.

No primeiro caso, como o sistema é composto de $2f + 1$ *acceptor*s, se $f + 1$ deles não haviam aceitado nenhuma requisição, os outros f não constituiriam uma maioria, e portanto nenhuma requisição poderia ter sido escolhida em i.

No segundo caso, se $f + 2$ *acceptor*s haviam aceitado requisições diferentes, sendo uma delas r, então mesmo que os restantes $f - 1$ *acceptor*s tenham aceitado r, a soma f não constituiria uma maioria, e portanto nenhuma requisição poderia ter sido escolhida em i.

Esses dois casos são a definição de $Free(i)$. Como corolário, para cada posição i que é $\neg Free(i)$ há um *acceptor* que aceitou uma requisição. Em PRaxos, todas as requisições escolhidas em um número de sequência $\neg Free(i)$ estão seguras de não ser substituídas por outra requisição, e, portanto, para cada número de sequência, apenas uma requisição proposta é escolhida. □

Lemma 3 *Propriedade da Não-Trivialidade: se uma requisição $r \in learned_p$ para cada processo $p \in \Pi$, então r foi proposto.*

A prova dessa propriedade é derivada dos lemas anteriores. Se uma requisição foi aprendida por um *learner*, então ela foi escolhida, e se uma requisição foi escolhida, então ela foi proposta. Suponha-se que a sequência de requisições aprendidas do processo p é $learned_p$, então, se há um processo p e uma requisição r, onde a requisição $r \in learned_p$, então q já havia sido proposto. □

Lemma 4 *Propriedade de Acordo: Se dois processos p e q em Π aprenderam a requisição r no número de sequência i, então $r = \texttt{learned}_p[i] = \texttt{learned}_q[i]$.*

O líder é o primeiro a aprender a requisição, e os demais processos a aprendem recebendo mensagens *Learn* do líder.

Como a requisição r havia sido escolhida em um número de sequência i, então ela é parte da seção *Processed* e está segura de não mudar de número de sequência. Portanto cada processo que aprende r o aprenderá em i: $\texttt{learned}_p[i] = r$. □

Lemma 5 *Propriedade de Ordem de Prioridade: Requisições prontas são escolhidas em ordem segundo sua prioridade.*

A ordem em que as requisições são executadas e aceitas pelos *acceptor*s é definida pelo líder. Cada vez que uma inversão de prioridade ocorre devido a uma requisição de alta prioridade, o líder reorganiza a seção *Ready* da sequência, onde os números de sequência são $Free(i)$, incluindo as requisições recebidas, na ordem apropriada. Cada processo *acceptor* p executa e aceita requisições uma de cada vez, a partir do $p.S[0]$ em diante. Desse modo, se uma requisição r foi escolhida em i, então $p.S[i]$ havia sido executado e aceitado por $f+1$ *acceptor*s, e antes disso, $p.S[i-1]$, e assim por diante, até o caso $p.S[0]$. Portanto, requisições prontas são escolhidas na ordem segundo a sua prioridade.

Contudo, nem todas as posições $\neg Free(i)$ são $Chosen(i)$. Isso significa que pode haver requisições *Ready* na seção *Chosen* da sequência que são executadas e escolhidas antes de requisições prontas de maior prioridade na seção *Ready*. Este caso pode acontecer se o líder acumula uma maioria de exatos $f+1$ *acceptor*s, onde um deles havia aceitado uma requisição r na posição i. A posição i não é qualificada como $Free(i)$, mas uma

requisição r pode não ter sido escolhida, porque pode ou não pode ter sido aceita por todos os f *acceptor*s que não participaram da maioria, seja devido a perdas ou atrasos de mensagens. PRaxos trata essas requisições como se fossem escolhidas, porque de outra forma múltiplas requisições poderiam ser escolhidas na mesma posição, o que quebra um dos requisitos de segurança.

A Figura 9 ilustra esse problema. Para o líder, δ e γ estão nessa situação. Ambas têm apenas um aceite, mas na verdade, δ está escolhido, e γ não. O líder não pode considerá-los $Free$, porque então δ causaria inversão de prioridade e mudaria de posição, o que não deveria acontecer para uma requisição escolhida. Consequentemente, o líder as considera $Chosen$, mesmo que isso leve γ a causar inversão de prioridade.

Se o sistema se comportasse de maneira síncrona, isto é, com um limite de tempo para a entrega de mensagens, então todas as réplicas corretas receberiam as mensagens $Propose$ do líder e responderiam com $Promise$, entregando as suas sequências com êxito. Nesse caso, todas as posições $Chosen(i)$ seriam $\neg Free(i)$, e requisições prontas são escolhidas na ordem segundo as suas prioridades. □

A correção do algoritmo vem dos lemas 3, 4 e 5.

4.2.3 Limitações

Apesar do Praxos ser um algoritmo muito conhecido, ele também tem a fama de ser difícil de entender devido às sutilezas do seu funcionamento e da escolha de expressá-lo em duas partes: o protocolo do Sínodo e o algoritmo Paxos, aquela para resolver o consenso de uma variável, e este para resolver o consenso de uma sequência de variáveis. Devido a isso, essas partes também são conhecidas como Single-Paxos e Multi-Paxos, e a separação do algoritmo entre as duas partes, focando justamente no Single-Paxos, é razão de dificuldades de entendimento. Além dos problemas de sutileza e de apresentação, Paxos não possui uma detalhada especificação prática para a sua implementação. Dentre os aspectos de que carece incluem-se o mecanismo de eleição de líder, de manutenção de sequências, troca de membros, detecção da falha de réplicas e a compactação de sequências. Todas essas características são oferecidas pelo Raft [Ongaro e Ousterhout 2014].

Raft é um algoritmo de consenso para máquinas de estado replicadas, assim como o Paxos, desenvolvida para ser uma alternativa mais simples, que possa ser não apenas mais facilmente ensinada, como desenvolvida. Como o Raft é um algoritmo muito mais prático de implementar, optou-se por desenvolver o PRaft, o algoritmo para resolver o PB-SMR baseado nesse algoritmo, e implementá-lo para fazer os testes comparativos de desempenho.

4.3 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 4.

Algoritmo 4 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 5 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.

O Algoritmo 6 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.

Algoritmo 5 Líder l recebe requisição r de cliente.

```
1: i ← posição em que r deve ser inserido no log
2: if e ≥ 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 ← 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 ← i
14:    end if
15: end for
16: for Para cada processo p do
17:    if executed_index_p ≥ i then
18:       executed_index_p ← i − 1
19:    end if
20: end for
```

Algoritmo 6 Determina a posição em que a requisição r é inserida.

```
1: for i ← len(log) − 1; i > commit_index; i ← i − 1 do
2:    if P(log[i]) ≥ P(r) then
3:       return i + 1
4:    end if
5: end for
6: return commit_index + 1
```

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

O Algoritmo 7 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 8 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 secção de índice

Algoritmo 7 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**

$prev\_index + 1$ em diante. Ocorre inversão de prioridade se o seguidor estiver executando alguma requisição dessa seção (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 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).

Algoritmo 8 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**

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 14.

O Algoritmo 9 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 10 e 11 descrevem o que acontece quando uma máquina de estado termina a execução de uma requisição. Em qualquer processo, seja líder ou seguidor, a

Algoritmo 9 Líder l recebe resposta res do seguidor s para o AppendEntry

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

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 **??** o seguidor envia nessa mensagem a variável *exec*, isto é, o índice da requisição que ele acabara de executar (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).

Algoritmo 10 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: $\quad execution\_index_l \leftarrow e - 1$
5: $\quad$ Atualiza o *commit_index*
6: **else**
7: $\quad$ Envia ao líder a mensagem de fim de execução da requisição em $exec \leftarrow e - 1$.
8: **end if**

Algoritmo 11 O líder recebe do seguidor s a mensagem msg de fim de execução

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

O Algoritmo 12 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.

4.4 ANÁLISE COMPARATIVA

Esta seção avalia PRaft e o PRaxos e os compara com os trabalhos relacionados. Apesar dos trabalhos mencionados tratarem do problema de inversão de prioridades, eles

Algoritmo 12 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: $\quad commit\_index \leftarrow index$
6: **end if**

dependem de premissas fortes. A Tabela 3 mostra a comparação entre os algoritmos estudados e os algoritmos desenvolvidos.

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 falhas nem troca de líder, em que uma mensagem é enviada, concordada, e entregue, sem se atrasar devido a inversão de prioridades, perda de mensagens ou suspeitas de falta nos algoritmos que usam detectores de faltas.

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 $n-1$ *AppendEntry*s seguintes que informam 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 $O(n)$.

PRaxos segue o mesmo número de passos de comunicação que o Paxos. Semelhantemente ao PRaft, as mensagens trocadas a cada mudança de líder, *Propose* e *Promise*, não são contadas, mas as que se repetem a cada requisição: $f-1$ *Accept*s e $2(f-1)$ *Learn*s. Estas somam $3(n-1)$ mensagens, onde n é o total de processos, e a complexidade também é linear $O(n)$.

Os algoritmos propostos neste trabalho têm a menor complexidade de mensagens entre os algoritmos relacionados.

Tabela 3: Comparação com os 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	$(2n^3+7n^2+5n-2)/2$	$O(n^3)$
Paxos	assíncrono	3	$3(n-1)$	$O(n)$
Raft	assíncrono	3	$3(n-1)$	$O(n)$
PRaxos	*eventual synchronous*	3	$3(n-1)$	$O(n)$
PRaft	*eventual synchronous*	3	$4(n-1)$	$O(n)$

4.5 CONSIDERAÇÕES FINAIS

Neste capítulo foram apresentados o modelo de sistemas, a descrição dos algoritmos de consenso para máquinas de estado replicadas baseado em prioridades: o PRaxos e o PRaft, e a comparação entre eles e os algoritmos relacionados.

De acordo com os objetivos desse trabalho foram apresentados algoritmos para o consenso em sistemas PB-SMR, capazes de tolerar faltas de *crash*, operando em complexidade de mensagens trocadas linear. Os algoritmos foram apresentados de maneira teórica, tendo o PRaxos uma seção de provas. No capítulo seguinte será apresentada a análise experimental do algoritmo PRaft.

5 AVALIAÇÃO EXPERIMENTAL

Neste capítulo é descrita uma avaliação experimental realizada a fim de validar a proposta apresentada nessa dissertação.

A avaliação experimental tem por objetivo elucidar algumas suposições descritas a seguir. Ao se utilizar um algoritmo baseado em prioridades, as requisições de alta prioridade executam antes das requisições de baixa prioridade. Por isso, requisições de alta prioridade teriam tempos de resposta mais curtos que requisições de baixa prioridade. Como algoritmos de consenso comuns (por exemplo, Paxos e Raft) geralmente não lidam com prioridades, as requisições são executadas na ordem em que chegam ao sistema, portanto o tempo de resposta é em média o mesmo para todas as requisições; essa afirmação estrutura a primeira hipótese a ser verificada nesta avaliação. A segunda hipótese que a avaliação busca confirmar é que o tempo de resposta das requisições de alta prioridade é menor no PRaft do que o tempo de resposta médio das requisições no Raft.

Enunciando as suposições apresentadas de forma sintetizada, pode-se declarar as hipóteses a serem avaliadas:

Hipótese 1: Em um sistema que considere prioridades, requisições com alta prioridade apresentam uma baixa latência em relação à latência percebida pelas requisições de baixa prioridade.

Hipótese 2: requisições de alta prioridade possuirão latência média menor do que a latência média de requisições que transitam em um sistema que não considera prioridades na execução de requisições.

Foi conduzido um experimento com o objetivo de descobrir a validade das hipóteses apresentadas. O experimento está descrito na próxima seção.

5.1 EXPERIMENTO

Para a realização do experimento, é necessária a implementação de um algoritmo que considere prioridade nas requisições. Diante das opções apresentadas na proposta, há uma razão de se implementar o PRaft e não o PRaxos para fazer as avaliações experimentais. O Paxos é um algoritmo complexo e pouco prático. Faltam especificações detalhadas para a sua implementação, o que faz com que um implementação do Paxos fique bem distante da sua especificação original. Implementar o Raft, contudo, é muito mais simples. O Raft

especifica detalhes como o mecanismo de eleição de líder, o funcionamento da sequência de requisições, um sistema de troca de mensagens que torna mais simples a verificação de faltas e mecanismo de atualização de réplicas que precisam conhecer as requisições confirmadas, como um novo líder por exemplo, ou uma réplica recuperada depois de uma falha. Devido a essas facilidades optou-se por implementar o Raft ao invés do Paxos, e adaptá-lo para tratar prioridades, isto é, transformá-lo em PRaft. Por causa disso o algoritmo PRaxos foi desenvolvido e demonstrado por provas informais, mas a implementação prática e testes de comparação foram feitas como PRaft.

A fim de avaliar as Hipóteses 1 e 2, foram implementados os algoritmos PRaft (descrito no capítulo da proposta) e Raft (descrito no capítulo de conceitos). Com a execução do PRaft será possível avaliar a Hipótese 1, e com a execução do PRaft e do Raft será possível avaliar a Hipótese 2.

Os protótipos dos algoritmos PRaft e Raft foram desenvolvidos na linguagem Python 1.3; a comunicação de rede entre o cliente e réplicas foi projetada para utilizar pacotes UDP. Os programas foram executados em quatro computadores, sendo um computador utilizado como cliente e três computadores utilizados como réplicas. Esta configuração permitiu ao sistema tolerar uma falta de parada, obedecida a relação $n = 2f + 1$, onde $f = 1$ e $n = 3$; f refere-se ao número de faltas toleradas e n refere-se ao número de réplicas no sistema (detalhes desta relação estão no capítulo de conceitos, seção 2.2.3). A execução do experimento caracterizou-se pelo acesso simultâneo de 19 clientes enviando 100 requisições ao sistema replicado. Os 19 clientes foram simulados com a execução de múltiplas *threads*. Um cliente só pode enviar um próximo pedido após receber a resposta do pedido anterior. A cada requisição é associada uma prioridade aleatória entre 0 e 10, e assim assume-se que para cada nível de prioridade foram executados aproximadamente o mesmo número de requisições. O tempo de execução é fixo em 1 segundo. As requisições são compostas pela prioridade da requisição (número), um comando (1 caracter) e o número de sequência, que representa o identificador da requisição. As respostas contém apenas 1 caracter.

Todos os computadores utilizados no experimento possuem configuração Intel i7 3.5Ghz, QuadCore, cache L3 8MB, 12GB RAM, 1TB HD 7200 RPM. Uma rede ethernet 10/100MBits isolada de tráfego externo conectou os computadores durante a realização dos experimentos. Os computadores possuem o sistema operacional Debian 7.4 Wheeze, 64 bits, com kernel versão 3.2.54-2.

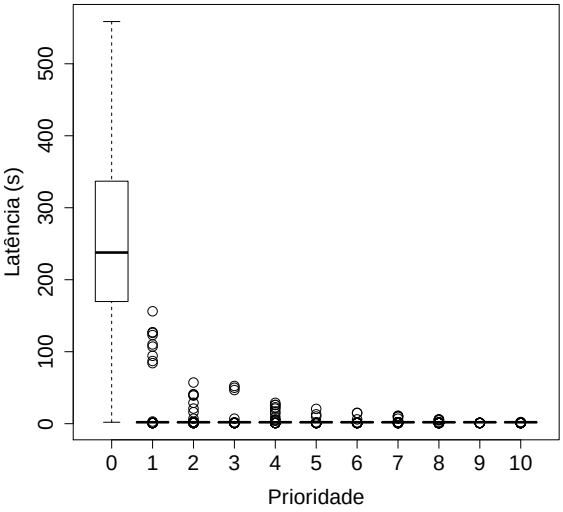

Figura 10: Latência do PRaft, em função das prioridades.

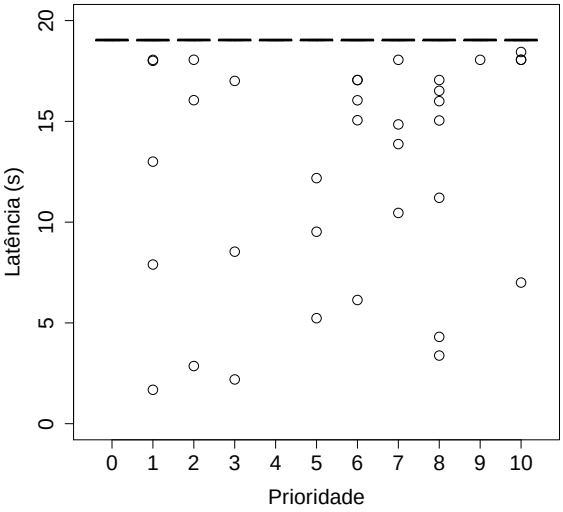

Figura 11: Raft: latências estáveis, poucos pontos discrepantes.

5.2 RESULTADOS

A Figura 10 mostra latências do PRaft, de acordo com os tipos de prioridade. Pode-se observar que as requisições de prioridade alta são executadas mais rápido em prejuízo das requisições de baixa prioridade, formando uma leve curva decrescente até a maior prioridade. A Tabela 4 mostra as latências médias com os respectivos desvios-padrão. Conforme pode-se observar, a latência média de todas as requisições que possuem prioridade maior que zero é inferior à latência média das requisições que não possuem prioridade (ou possuem prioridade igual a zero). Assim, afirma-se que a Hipótese 1 é **verdadeira**.

Tabela 4: PRaft: prioridades (P), latências médias (M) e desvios-padrão (DP).

P	0	1	2	3	4	5	6	7	8	9	10
M	256.29	15.43	3.34	10.20	2.77	2.15	2.07	9.78	1.95	1.91	1.89
DP	134.63	101.53	7.14	97.33	4.01	1.83	1.45	97.57	0.64	0.30	0.32

Uma observação relevante é que a aparente curva da Figura 10 é delineada por requisições discrepantes. O gráfico de caixas foi usado pois evidencia os pontos discrepantes. Há 3 pontos discrepantes que foram omitidos para não distorcer o gráfico. Estes pontos são relativos a requisições de prioridades 1, 3 e 7, e possuem os valores 1285,59s, 1276,57s e 1251,56s, respectivamente.

A execução do Raft foi estável, conforme mostra a Figura 11. Poucos pontos discrepantes alcançaram latência menor que a latência média, que foi igual a 18.91 ms e desvio padrão de 1.14 ms. O Raft não distingue as requisições por prioridade. Sendo que a latência média das requisições com prioridade (exibidas na Tabela 4) manteve-se abaixo da média das requisições executadas no Raft, pode-se concluir que as requisições de alta

prioridade executam mais rápido no PRaft do que Raft. Adicionalmente, a Figura 12 compara as latências medidas na execução do PRaft e do Raft. As latências do Raft tiveram suas prioridades deslocadas em meio ponto apenas para facilitar a visualização simultânea no gráfico. Percebe-se que o Raft segue uma média independente da prioridade, enquanto o PRaft alcança menor latência na medida em que a prioridade aumenta. Isso demonstra que o PRaft é adequado para processamento de requisições com prioridade. Pode-se então afirmar que a Hipótese 2 é **verdadeira**.

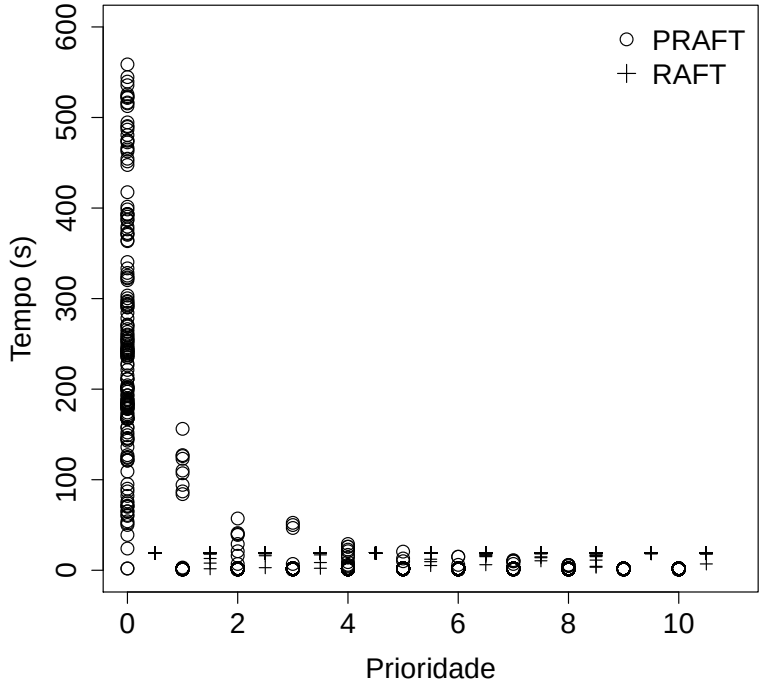

Figura 12: Comparação entre latências das execuções no PRaft e RAFT.

5.2.1 Discussões adicionais

Apesar de a latência de requisições com prioridade ser menor que a latência de requisições sem prioridade (ou com prioridade igual a zero), a sequência de médias de latências apresentada na Tabela 4 apresenta variabilidade em relação ao crescimento da prioridade. Os três pontos discrepantes mencionados anteriormente influenciam significativamente a média. Para averiguar essa influência, uma nova tabulação das médias foi feita, removendo-se os três pontos discrepantes dos dados. As novas médias estão dispostas na Tabela 5.

Tabela 5: PRaft: prioridades (P), latências médias (M) e desvios-padrão (DP), após exclusão de pontos discrepantes.

P	0	1	2	3	4	5	6	7	8	9	10
M	256.29	**7.87**	3.34	**2.80**	2.77	2.15	2.07	**2.16**	1.95	1.91	1.89
DP	134.63	**25.55**	7.14	**6.31**	4.02	1.83	1.45	**1.39**	0.64	0.30	0.32

As médias alteradas em função da remoção dos pontos discrepantes estão em destaque na Tabela 5. A latência média agora se estabelece de forma gradual em função da prioridade. Mediante a melhor relação entre latências médias e prioridades, é oportuno investigar a utilidade em se utilizar diversos níveis de prioridade. Assim, foi feita uma seleção de dados para incluir apenas as prioridades maiores que zero. Essa seleção de dados contemplou 1729 medições de latências. A Figura 13 apresenta as latências destas prioridades selecionadas. Os pontos em destaque são tão discrepantes tal que apenas traços foram exibidos onde estariam retângulos indicando quartis relativos à variação das latências.

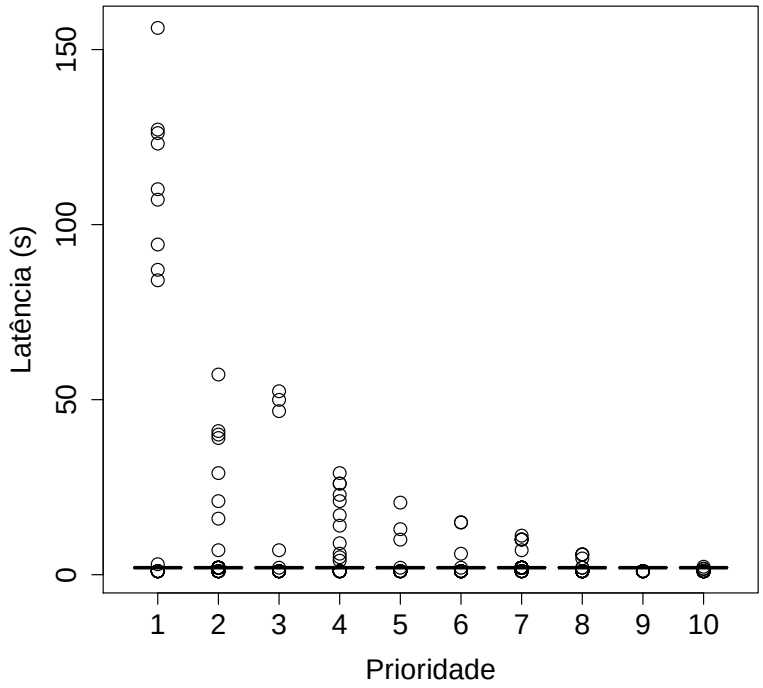

Figura 13: Latências de requisições com prioridades maior que zero.

É possível observar na Figura 13 que há uma pequena quantidade de pontos discrepantes; por exemplo, para a prioridade 1 podem ser identificados nove pontos com latência acima de 50s na figura. Para confirmar esta conjectura, pode-se visualizar os dados sem a presença dos pontos discrepantes: a Figura 14 mostra as latências desconsiderando os pontos discrepantes. É possível ver que as medianas das requisições são bastante próximas, independente das prioridades.

Aprofundando a averiguação sobre os níveis de prioridades, foi feita uma subseleção de dados, a partir da seleção anterior, especificando um critério de latência menor que 2.1 ms, visto que a faixa de latências mostrada na Figura 14 alcança no máximo 2.004 ms. A subseleção registrou 1680 latências. A Figura 15 revela então que as latências predominantemente mantiveram-se na faixa de 2 segundos, tendo ainda algumas latências na faixa de 1 segundo e duas ocorrências de latências entre 1 e 2 segundos.

Conclui-se assim que, dentre as requisições que utilizam prioridade, a diminuição

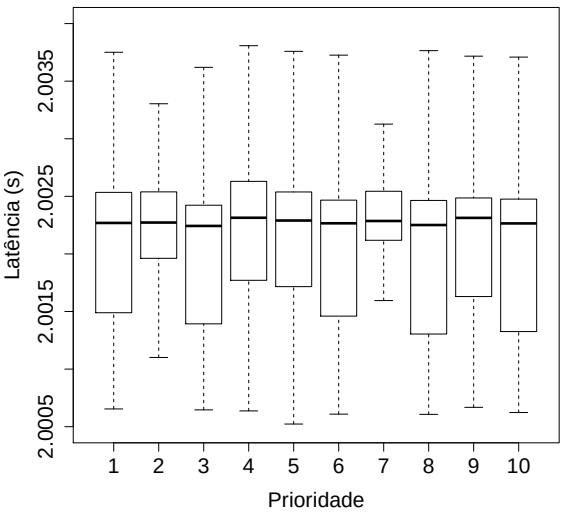

Figura 14: Latências de requisições com prioridade, sem pontos discrepantes.

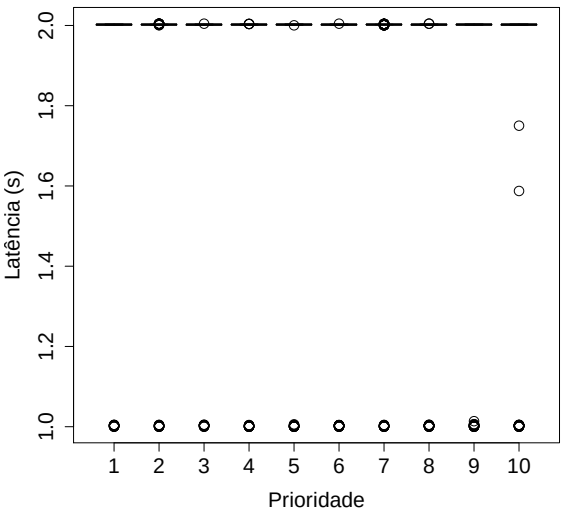

Figura 15: Latências do PRaft sem nenhum ponto discrepante.

da latência em função de uma maior prioridade é apenas resultado de uma minoria de requisições (pontos discrepantes). Portanto, a divisão de prioridades em vários níveis não oferece vantagens significativas, em comparação aos resultados da execução com ou sem prioridade. Possivelmente, uma categorização apenas em requisições **com** ou **sem** prioridade alcancaria resultados tão satisfatórios quanto os mostrados nesta seção.

5.3 CONSIDERAÇÕES DO CAPÍTULO

Este capítulo apresentou uma avaliação experimental do uso de prioridades em protocolos de ordenação. O PRaft, proposto nessa dissertação, foi implementado e os resultados mostraram que requisições com prioridade são executadas com menor latência do que requisições sem prioridade. Além disso, O PRaft foi comparado ao Raft [Ongaro e Ousterhout 2014], e foi possível observar que requisições com prioridade são executadas mais rapidamente no PRaft do que no Raft, visto que o Raft não trata prioridades em requisições.

6 CONCLUSÕES

Nesse trabalho foi apresentada uma proposta para resolver o problema PB-SMR, isto é, o problema de replicação de máquina de estado tolerante a faltas baseado em prioridades, em que requisições de alta prioridade feitas para o sistema são executadas antes das requisições de baixa prioridade. Os trabalhos [Nakamura e Takizawa 1992], [Rodrigues, Veríssimo e Casimiro 1995] e [Wang et al. 2002] tratam esse problema, mas devido à sua dependência de serviços para resolver o problema de consenso, o número de passos do algoritmo passa a ter complexidade quadrática ou mesmo cúbica.

Visando a evolução do estado da arte em relação aos trabalhos apresentados e atender ao objetivo geral, estes objetivos específicos foram perseguidos:

1. Avaliação do estado da arte na área de consenso com prioridade;
2. Proposição de um algoritmo que adapte o Paxos para lidar com prioridades;
3. Proposição e desenvolvimento de um algoritmo que adapte o Raft para tratar prioridades;
4. Análise comparativa dos algoritmos propostos com os trabalhos relacionados;
5. Implementação e realização de uma avaliação experimental de um dos algoritmos propostos, comparando-o com o algoritmo original do qual ele foi derivado.

As soluções apresentadas, PRaxos e PRaft, resolvem o problema de máquina de estado replicada baseada em prioridades em tempo linear. Ademais, os resultados dos testes que comparam o PRaft ao Raft mostram que à medida que a prioridade da requisição aumenta, menor é a média do seu tempo de resposta. A mediana, contudo, é quase a mesma para todas as prioridades diferentes de zero, e essas medianas são menores do que a mediana do tempo de resposta das requisições processadas pelo Raft. Isso demonstra que o PRaft é adequado para o processamento de requisições de prioridade.

6.1 VISÃO GERAL DO TRABALHO

Nessa seção é revisado o trabalho e de como foram atendidos os objetivos na seção anterior.

O capítulo 1 apresentou a contextualização do trabalho, o problema, os objetivos e as contribuições. O capítulo 2 introduziu os conceitos básicos de computação distribuída, tolerância a faltas, prioridades e tempo real que foram usados no restante do trabalho. O

capítulo 3 mostrou os trabalhos relacionados ao problema de consenso com prioridades, e a comparação destes trabalhos com a solução proposta nessa dissertação. O capítulo 4 descreveu os algoritmos desenvolvidos para o problema de replicação de máquinas de estado baseados em prioridade: PRaxos e PRaft, algoritmos que executam com número de mensagens de complexidade linear, com uma prova não-formal da correção do PRaxos. O capítulo 5 apresentou os testes de comparação e os resultados entre o PRaft e o Raft, onde foi demonstrado que o PRaft responde a requisições de alta prioridade em menor tempo que a média das requisições respondidas pelo Raft.

6.2 CONTRIBUIÇÕES

A partir dos objetivos, as contribuições deste trabalho podem ser enumeradas como segue:

1. Um algoritmo de replicação de máquinas de estado baseado em prioridade para resolver o problema de consenso levando em conta requisições com prioridades: PRaxos. Um algoritmo baseado no Paxos.

2. O algoritmo de replicação de máquinas de estado baseado em prioridade para resolver o problema de consenso levando em conta requisições com prioridades: PRaft. Um algoritmo baseado no Raft.

3. A implementação do PRaft e uma análise comparativa com o Raft.

Com o objetivo de divulgar os resultados obtidos à comunidade científica, e, principalmente, submetê-los à sua avaliação, foram produzidos artigos e submetidos a eventos na área de computação distribuída e tolerância a faltas. As revisões contribuíram para o amadurecimento deste trabalho e na sua atual apresentação. Um artigo foi aceito para publicação:

1. PINHO, Paulo; RECH, Luciana; LUNG, Lau Cheuk; CORREIA, Miguel; CAMARGOS, Lasaro. Priority-Based State Machine Replication with PRaxos. In: The 30-th IEEE International Conference on Advanced Information Networking and Applications (AINA-2016), Crans-Montana, Switzerland, 2016. Qualis *A2*.

2. PINHO, Paulo; RECH, Luciana; LUNG, Lau Cheuk; CORREIA, Miguel; CAMARGOS, Lasaro. Replicação de Máquina de Estado Baseada em Prioridade com PRaft. In: Simpósio Brasileiro de Redes de Computadores (SBRC). Salvador, Brasil, 2016. Qualis *B2*

6.3 TRABALHOS FUTUROS

Entre possíveis trabalhos futuros estão a adaptação de algoritmos tolerantes a faltas bizantinas para lidar com prioridades, a implementação e o teste de outras práticas de escalonamento, e a comparação de desempenho dos algoritmos desenvolvidos com os algoritmos dos trabalhos relacionados.

Há desafios para implantar em BFT a solução para tratar prioridades. Para os algoritmos tolerantes a faltas de *crash* o líder é responsável pela organização da sequência de requisições, como se pode ver na seção de proposta. A dificuldade é que o líder do BFT pode tratar incorretamente a solução da prioridade, manifestando um comportamento diferente do esperado. A detecção desse comportamento é difícil.

REFERÊNCIAS

AIYER, A. S. et al. BAR fault tolerance for cooperative services. In: *Proceedings of the 20th ACM Symposium on Operating Systems Principles*. [S.l.: s.n.], 2005.

ALPERN, B.; SCHNEIDER, F. B. Defining liveness. *Information processing letters*, Elsevier, v. 21, n. 4, p. 181–185, 1985.

BESSANI, A. et al. DepSky: Dependable and secure storage in a cloud-of-clouds. In: *Proceedings of the 6th ACM SIGOPS/EuroSys European Systems Conference*. Saltzburg, Austria: ACM, New York, NY, 2011. p. 31–46.

BUYYA, R.; YEO, C. S.; VENUGOPAL, S. 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*. [S.l.: s.n.], 2008. p. 5–13.

CASTRO, M.; LISKOV, B. Practical Byzantine fault tolerance. In: *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*. [S.l.: s.n.], 1999. p. 173–186.

CASTRO, M.; LISKOV, B. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, ACM, New York, NY, USA, v. 20, n. 4, p. 398–461, nov. 2002. ISSN 0734-2071. <http://doi.acm.org/10.1145/571637.571640>.

CHANDRA, T. D.; TOUEG, S. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, v. 43, n. 2, p. 225–267, 1996.

CRISTIAN, F.; FETZER, C. The timed asynchronous system model. In: *Proceedings of the 28th IEEE International Symposium on Fault-Tolerant Computing*. [S.l.: s.n.], 1998. p. 140–149.

DWORK, C.; LYNCH, N.; STOCKMEYER, L. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, ACM, v. 35, n. 2, p. 288–323, 1988.

FETZER, C.; CRISTIAN, F. On the possibility of consensus in asynchronous systems. In: *Proceedings of the 1995 Pacific Rim International Symposium on Fault-Tolerant Systems*. [S.l.: s.n.], 1995. p. 86–91.

FISCHER, M. J.; LYNCH, N. A.; PATERSON, M. S. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, ACM, v. 32, n. 2, p. 374–382, 1985.

HADZILACOS, V.; TOUEG, S. Fault-tolerant broadcasts and related problems. In: MULLENDER, S. (Ed.). *Distributed Systems*. [S.l.]: ACM Press/Addison-Wesley, 1993. p. 97–145.

HURFIN, M. et al. A general framework to solve agreement problems. In: *Proceeding of the 18th International Symposium on Reliable Distributed Systems*. [S.l.: s.n.], 1999. p. 56–65.

KOTLA, R. et al. Zyzzyva: speculative Byzantine fault tolerance. In: *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*. [S.l.: s.n.], 2007. p. 45–58.

LAMPORT, L. The part-time parliament. *ACM Trans. Comput. Syst.*, ACM, New York, NY, USA, v. 16, n. 2, p. 133–169, maio 1998. ISSN 0734-2071. <http://doi.acm.org/10.1145/279227.279229>.

LAMPORT, L. Paxos made simple. *ACM Sigact News*, v. 32, n. 4, p. 18–25, 2001.

LAMPORT, L.; SHOSTAK, R.; PEASE, M. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, ACM, New York, NY, USA, v. 4, n. 3, p. 382–401, jul. 1982. ISSN 0164-0925. <http://doi.acm.org/10.1145/357172.357176>.

LOCKE, C. D. *Best-effort Decision-making for Real-time Scheduling*. Tese (Doutorado), 1986.

LUIZ, A. F.; LUNG, L. C.; CORREIA, M. Byzantine fault-tolerant transaction processing for replicated databases. In: *Proceedings of the 10th IEEE International Symposium on Network Computing and Applications*. [S.l.: s.n.], 2011. p. 83–90.

MIEDES, E.; MUNOZ-ESCOI, F. D. *Adding priorities to total order broadcast protocols*. [S.l.], October 2007.

NAKAMURA, A.; TAKIZAWA, M. Priority-based total and semi-total ordering broadcast protocols. In: *Proceedings of the 12th International Conference on Distributed Computing Systems*. [S.l.: s.n.], 1992. p. 178–185.

NAKAMURA, A.; TAKIZAWA, M. Starvation-prevented priority-based total ordering broadcast protocol on high-speed single channel network. In: IEEE. *High Performance Distributed Computing, 1993., Proceedings the 2nd International Symposium on*. [S.l.], 1993. p. 281–288.

OKI, B. M.; LISKOV, B. 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*. [S.l.: s.n.], 1988. p. 8–17.

ONGARO, D.; OUSTERHOUT, J. In search of an understandable consensus algorithm. In: *2014 USENIX Annual Technical Conference*. [S.l.: s.n.], 2014. p. 305–319.

ONGARO, D.; OUSTERHOUT, J. In search of an understandable consensus algorithm. In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, 2014. p. 305–319. ISBN 978-1-931971-10-2. <https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro>.

RODRIGUES, L.; VERÍSSIMO, P.; CASIMIRO, A. Priority-based totally ordered multicast. In: CITESEER. *3rd IFIP/IFAC workshop on Algorithms and Architectures for Real-Time Control (AARTC'95*. [S.l.], 1995.

SCHIPER, A.; RICCIARDI, A. Virtually-synchronous communication based on a weak failure suspector. In: *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing*. [S.l.: s.n.], 1993. p. 534–543.

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

VERISSIMO, P.; RODRIGUES, L. *Distributed Systems for System Architects*. [S.l.]: Kluwer Academic Publishers, 2001.

WANG, Y. et al. Solving the group priority inversion problem in a timed asynchronous system. *IEEE Transactions on Computers*, IEEE Computer Society, Los Alamitos, CA, USA, v. 51, n. 8, p. 900–915, 2002. ISSN 0018-9340.