Implementação e Avaliação do Protocolo de Difusão Atômica Rápida à Despeito de Colisões

Rodrigo Queiroz Saramago

Universidade Federal de Uberlândia
Faculdade de Computação
Programa de Pós-Graduação em Ciência da Computação

Uberlândia
2016

Rodrigo Queiroz Saramago

Implementação e Avaliação do Protocolo de Difusão Atômica Rápida à Despeito de Colisões

Dissertação de mestrado apresentada ao Programa de Pós-graduação da Faculdade de Computação da Universidade Federal de Uberlândia como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Ciência da Computação

Orientador: Lásaro Jonas Camargos

Uberlândia
2016

Dados Internacionais de Catalogação na Publicação (CIP)
Sistema de Bibliotecas da UFU, MG, Brasil.

S243i
2016

Saramago, Rodrigo Queiroz, 1986-
Implementação e avaliação do protocolo de difusão atômica rápida à despeito de colisões / Rodrigo Queiroz Saramago. - 2016.
110 f. : il.

Orientador: Lásaro Jonas Camargos.
Dissertação (mestrado) - Universidade Federal de Uberlândia, Programa de Pós-Graduação em Ciência da Computação.
Disponível em: http://dx.doi.org/10.14393/ufu.di.2018.1147
Inclui bibliografia.

1. Computação - Teses. 2. Sistemas distribuidos - Protocolos - Teses. 3. Algoritmos - Teses. I. Camargos, Lásaro Jonas. II. Universidade Federal de Uberlândia. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDU: 681.3

Maria Salete de Freitas Pinheiro – CRB6/1262

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Os abaixo assinados, por meio deste, certificam que leram e recomendam para a Faculdade de Computação a aceitação da dissertação intitulada "**Implementação e Avaliação do Protocolo de Difusão Atômica Rápida à Despeito de Colisões**" por **Rodrigo Queiroz Saramago** como parte dos requisitos exigidos para a obtenção do título de **Mestre em Ciência da Computação.**

Uberlândia,08 de Setembro de 2016

Orientador: _____

Prof. Dr. Lásaro Jonas Camargos
Universidade Federal de Uberlândia

Banca Examinadora:

_____

Prof. Dr. Eduardo Adilio Pelinson Alchieri
Universidade de Brasília

_____

Prof. Dr. Luís Fernando Faina
Universidade Federal de Uberlândia

_____

Prof. Dr. Gustavo Maciel Dias Vieira
Universidade Federal de São Carlos

Este trabalho é dedicado às crianças adultas que,
quando pequenas, sonharam em se tornar cientistas.

Agradecimentos

Gostaria de agradecer a todos que direta ou indiretamente contribuíram na conclusão deste trabalho. Principalmente ao meu orientador e amigo Lásaro pela paciência e confiança, cujo apoio e conselhos foram de grande valia no decorrer deste trabalho, e aos meus amigos Tuanir e Enrique pelas discussões, nem sempre produtivas, mas necessárias.

Agradeço também a Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pelo apoio financeiro, indispensável.

Finalmente, agradeço a minha família por acreditarem no meu potencial e me apoiarem nas minhas decisões.

"When you see a good move, look for a better one."
Emanuel Lasker

Resumo

Replicação de Máquinas de Estados, uma técnica comum para se alcançar tolerância a falhas, pode ser implementada por meio de primitivas de Difusão Atômica (*Atomic Broadcast*). Difusão Atômica, por sua vez, é comumente implementada através de algoritmos de *Consensus*: utilizando infinitas instâncias de *Consensus* totalmente ordenadas decide-se por uma sequência de comandos a serem executados na máquina de estados replicada. Esta abordagem tem a desvantagem de forçar as propostas não decididas (comandos não entregues) a serem repropostas em novas instâncias, atrasando sua execução. Algoritmos que evitam tais problemas são denominados *collision-fast* e apresentam uma latência ótima constituída de dois passos de comunicação. Os existentes, contudo, requerem um certo grau de sincronismo (Clock-RSM), ou não tratam falhas de forma eficiente (Mencius) ou ainda não foram avaliados experimentalmente (CFABCAST). Este trabalho objetiva primariamente a implementação do algoritmo Collision-fast Atomic Broadcast (CFABCAST), bem como uma avaliação de desempenho em relação ao modelo clássico de replicação de máquinas de estado baseado no *Paxos*, e a outro trabalho denominado *Multi Ring Paxos*. Além disso, este trabalho tem como objetivos adicionais, melhorar a eficiência do protocolo em sistemas que permitam execução especulativa e torná-lo resiliente a falhas bizantinas. Por fim, conjecturamos ser impossível existir um protocolo *Collision-fast* que tolere falhas bizantinas.

Palavras-chave: Consenso Distribuído, Consenso Bizantino, Difusão Atômica, Paxos, Multi Ring Paxos, M-Consensus, Collision-fast, Sistemas Distribuídos, Replicação de máquinas de estado.

Abstract

State Machine Replication is a common technique for achieving fault tolerance that can be implemented by Atomic Broadcast primitives. Atomic Broadcast is usually implemented by solving infinitely many instances of the well-known consensus problem. This approach has the disadvantage of forcing the concurrent broadcast of messages that have not yet been decided, causing them to be re-proposed in new instances, therefore delaying execution. Collision-fast algorithms, which deliver many messages within two message steps in good runs, exist, but either make assumptions that may be too restrictive; require a certain degree of clock synchronization among nodes; do not deal efficiently with failures or have not been experimentally evaluated. In this work we propose an architecture to implement the Collision-fast Atomic Broadcast algoritm as part of a distributed service, exploring the parallelism in today's machines, and also evaluating the performance of this protocol in a variety of scenarios, comparing it with other two protocols (Paxos and Multi-Ring Paxos). Moreover, this work aims at improving the protocol to allow speculative execution of delivered commands and make it resilient to Byzantine failures. Finally, we conjecture the impossibility of Byzantine failure tolerant Collision-fast protocols.

Keywords: Distributed Consensus, Byzantine Consensus, Atomic Broadcast, Paxos, Multi Ring Paxos, M-Consensus, Collision-fast, Dystributed systems, state machine replication.

Lista de ilustrações

Lista de tabelas

Lista de siglas

CAP Consistency, Availability and Partition tolerance

CAPES Coordenação de Aperfeiçoamento de Pessoal de Nível Superior

CFABCAST Collision-fast Atomic Broadcast

JVM Java Virtual Machine

Libmcad Multicast Adaptor Library

TLA+ Temporal Logic of Actions

WAN Wide Area Network

Lista de Algoritmos

Sumário

Capítulo 1

Introdução

1.1 Motivação

Replicação de máquinas de estados, do inglês *state machine replication*, é um método muito utilizado para se alcançar redundância e aumento de disponibilidade de serviços. A técnica consiste em processar em todas as réplicas de um serviço os mesmos comandos de forma determinista e em uma mesma ordem, consequentemente, levando-as a transições de estado consistentes (LAMPORT, 1978; LAMPSON, 1996)

O problema de Difusão Atômica, do inglês *Atomic Broadcast* consiste em, dado um conjunto de mensagens difundidas, entregar tais mensagens atomicamente, isto é, ou todos os processos recebem ou nenhum recebe, e em uma mesma ordem a todos os receptores não falhos (DÉFAGO; SCHIPER; URBÁN, 2003).

É fácil ver que dado um algoritmo de difusão atômica, é possível implementar a replicação de máquina de estados difundindo-se os comandos como mensagens para as réplicas. De fato, algoritmos de difusão atômica são comumente utilizados na implementação de replicação de máquinas de estado.

A Difusão Atômica é equivalente ao problema de Consenso Distribuído, do inglês *Distributed Consensus*, no qual um conjunto de processos decide por um dentre um conjunto de valores propostos (CHANDRA; TOUEG, 1995). Na redução do primeiro para o segundo problema, uma instância de consenso é usada para decidir cada mensagem a ser entregue. Utilizando-se infinitas instâncias de consenso e estabelecendo-se uma ordenação total das instâncias, define-se a ordem das mensagens entregues.

É conhecido que cada instância de consenso, em ambiente assíncrono e sujeito a falhas, tem latência mínima de dois passos de comunicação (LAMPORT, 2006b), e que diversos algoritmos atingem tal latência, e.g., *Fast Paxos* (LAMPORT, 2006a). Assim, dada a redução da difusão atômica ao consenso distribuído, esperaria-se que cada mensagem difundida pudesse ser entregue em dois passos de comunicação, mas este não é o caso na prática.

Algoritmos baseados em consenso, em geral, não conseguem garantir uma latência

ótima, de dois passos de comunicação, devido à ocorrência de propostas concorrentes, fenômeno este denominado *colisão*. Quando uma colisão acontece, no máximo uma das mensagens envolvidas será decidida –e entregue–, sendo que as demais devem ser entregues usando outras instâncias de consenso, o que soma às suas latências de entrega.

Algoritmos que tem latência de dois passos de comunicação na presença de colisões, conhecidos como *Collision-fast*, são raros na literatura e baseiam-se em algum artifício que lhes permitem contornar a colisão sem precisar de uma nova instância. Tais artifícios podem limitar o uso do algoritmo em aplicações reais. Dos algoritmos *Collision-fast* conhecidos, os de Lamport (LAMPORT, 2006b), restringem o número de processos que podem ser usados para a resolução do consenso –o que limita sua tolerância a falhas–, (MAO; JUNQUEIRA; MARZULLO, 2008) só é *collision-fast* até que uma falha aconteça, devendo ser reiniciado após uma ocorrência, e (DU et al., 2014) depende de relógios sincronizados para garantir a latência ótima.

O algoritmo *Collision-Fast Atomic Broadcast* é diferente dos demais existentes por usar uma abstração do problema semelhante ao consenso, mas mais genérica, denominada *M-Consensus* (SCHMIDT; CAMARGOS; PEDONE, 2007). Nesta variante do consenso, vários valores podem ser decididos em uma mesma instância. Assim, ainda que cada processo possa propor um único valor, ao final de uma instância de *M-Consensus*, várias propostas, possivelmente todas, serão decididas, o que se traduz na implementação da difusão atômica em várias mensagens sendo entregues ao final de cada instância.

Dado que o protocolo *Collision-Fast Atomic Broadcast* usa um algoritmo de *M-Consensus* com latência de dois passos de comunicação, o algoritmo é *collision-fast* (SCHMIDT; CAMARGOS; PEDONE, 2007). A particularidade do algoritmo *Collision-Fast Atomic Broadcast* resolver o problema de difusão atômica utilizando infinitas instâncias de *M-Consensus* não impõe penalidades como as outras abordagens que utilizam *Consensus*. Tal algoritmo, contudo, não foi validado experimentalmente, o que provavelmente impedirá ou retardará seu uso em aplicações reais.

Dada a difusão da técnica de replicação de máquinas de estados, o uso de um algoritmo de Difusão Atômica eficiente impacta diretamente no desempenho de toda uma cadeia de aplicações. Assim, é essencial conhecer qual algoritmo de difusão atômica se presta melhor à tarefa.

1.2 Objetivos e Desafios da Pesquisa

Este trabalho tem como objetivo principal caracterizar, teórica e experimentalmente, protocolos de Difusão Atômica *Collision-fast*, ressaltando suas vantagens, desvantagens e em que cenários se aplicam.

O objetivo principal pode ser pormenorizado nos seguintes objetivos secundários deste trabalho:

- ❏ implementar o algoritmo *Collision-fast Atomic Broadcast* e validá-lo com o resultado teórico apresentado em (SCHMIDT; CAMARGOS; PEDONE, 2007);
- ❏ definir e implementar um conjunto de testes de desempenho a ser aplicado a algoritmos de difusão atômica.
- ❏ comparar algoritmos existentes sob a óptica dos testes desenvolvidos.

1.3 Contribuições

Neste trabalho implementamos o protocolo de *Atomic broadcast* proposto em (SCHMIDT; CAMARGOS; PEDONE, 2007) com base na sua especificação escrita em Temporal Logic of Actions (TLA+). Nós avaliamos nossa implementação utilizando um ambiente de testes oferecido pelo Emulab e observamos que diferente do *Paxos*, o *Collision-fast Paxos* não apresenta degradação significativa de performance com o aumento do número de propostas concorrentes, possibilitando que mais de um valor seja decidido por instância de Consenso.

Assim como em (ZHAO, 2015), no decorrer deste trabalho mostramos que, na prática, é necessária a adição de vários mecanismos que não estavam especificados no protocolo, mas que são de vital importância para o funcionamento do protocolo em ambientes reais. Desta forma, acabamos por desenvolver uma *framework* de testes de protocolos de acordo.

Ademais, uma versão inicial da arquitetura de nossa implementação e método de avaliação foi publicado como *fast-abstract* na *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) 2015* (SARAMAGO; CAMARGOS, 2015).

Além da implementação e avaliação, propomos uma extensão do protocolo *Collision-fast Paxos* que tolera falhas bizantinas utilizando o método de assinatura digital para garantir a autenticidade das mensagens. Apesar de ser uma extensão de um protocolo *collision-fast*, o protocolo resultante não é *collision-fast*. De fato, conjecturamos ser impossível existir um protocolo *collision-fast* que tolere falhas bizantinas nos mesmos termos que um protocolo não bizantino.

Também alteramos o protocolo para permitir a execução especulativa de comandos quando suportado pela aplicação cliente. O algoritmo proposto foi implementado no nosso *framework* e possibilitou o desenvolvimento de várias políticas de entrega de mensagens.

Os experimentos conduzidos no Emulab, simulando uma rede de longa distância – Wide Area Network (WAN)–, mostraram melhor desempenho do *Collision-fast Paxos* em comparação com o *Paxos*, quando utilizados na replicação de máquinas de estado, como indicado pela literatura (SCHMIDT; CAMARGOS; PEDONE, 2014). E pelo que nos concerne essa é a única implementação existente de tal protocolo disponibilizada publicamente como um projeto *open source*[1].

[1] <https://github.com/r0qs/cfabcast>

1.4 Organização da Dissertação

O restante desta dissertação é organizado da seguinte forma. No Capítulo 2 apresentamos a teoria na qual esse trabalho se fundamenta, enfatizando a importância de protocolos de acordo em ambientes distribuídos. Também apresentamos nesse capítulo trabalhos cujos objetivos são similares ao nosso e qual a vantagem do nosso diante deles. No Capítulo 3 descrevemos a arquitetura da nossa implementação, justificando as nossas escolhas de projeto do sistema e explicando o funcionamento da nossa implementação, baseada na especificação apresentada em TLA+ presentes em (SCHMIDT; CAMARGOS; PEDONE, 2007). No Capítulo 4 apresentamos o método que utilizamos para avaliar nossa implementação, bem como os resultados obtidos. Em seguida, no Capítulo 5 propomos uma extensão do modelo de tolerância a falhas do protocolo original, permitindo que o este suporte falhas bizantinas. Por fim, concluímos este trabalho no Capítulo 6, discutindo possibilidades de melhorias e trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Neste capítulo discutiremos conceitos que servem de fundamentação teórica para nosso trabalho bem como diversos trabalhos relacionados ao nosso.

2.1 Modelo do sistema

Como em (SCHMIDT; CAMARGOS; PEDONE, 2007), neste trabalho considerou-se um sistema composto por um conjunto finito de agentes (entidades que executam alguma tarefa computacional, e.g. processos, *threads*, atores, etc) que se comunicam por troca de mensagens em um modelo computacional parcialmente síncrono, no qual não há limite no tempo de transmissão das mensagens ou ações executadas pelos agentes.

Além disso, considera-se que mensagens possam ser perdidas ou duplicadas mas não corrompidas, e que se repetidamente reenviadas de um processo correto para outro, são entregues em algum momento. E finalmente, falhas são do tipo *crash-stop*, isto é, processos podem parar de funcionar mas nunca executam ações maliciosas, ou seja, falhas Bizantinas.

2.2 Consenso Distribuído

O problema de Consenso é a base para muitos outros problemas em sistemas distribuídos, como Difusão Atômica. De fato, tais problemas são equivalentes, sendo possível reduzir um problema ao outro (CHANDRA; TOUEG, 1995). No consenso, processos devem propor valores e, ao final da instância, acordar em um dentre os vários valores propostos; tal acordo é chamado de *decisão* e é aprendida por todos os processos corretos interessados. Entende-se por aprender a capacidade de um processo receber e utilizar uma informação recebida, por exemplo, repassando uma decisão para uma camada de aplicação. Um processo é dito correto se ele não falha durante a execução do algoritmo.

Consenso pode ser modelado em termos de um conjunto de papéis desempenhados por agentes no protocolo (LAMPORT, 1998). Um agente pode executar um ou mais dos

seguintes papéis:

- *proposers* – podem propor um valor;
- *acceptors* – cooperam na escolha de um valor como decisão; e,
- *learners* – podem aprender qual valor foi escolhido.

Além disso, algumas definições são necessárias para melhor entendimento do problema:

- *Leader* ou *Coordinator:* um tipo de *proposer* responsável por iniciar cada rodada do protocolo e tem como função principal garantir a terminação do protocolo.
- *Quorums:* subconjuntos do conjunto de *acceptors* que estabelecem uma condição necessária para se alcançar um consenso. Tais conjuntos são definidos de modo que quaisquer dois *quorums* possuam uma intersecção não vazia, ou seja, possuem pelo menos um elemento (*acceptor*) em comum . Uma forma simples de garantir tal propriedade é definir um *quorum* como uma maioria dos *acceptors*.

Formalmente, Consenso é definido pelas seguintes propriedades:

a) *não trivialidade:* qualquer valor aprendido deve ter sido proposto por algum agente no sistema;

b) *estabilidade:* um *learner* pode aprender no máximo um valor proposto;

c) *consistência:* dois *learners* diferentes não podem aprender valores diferentes.

Além das propriedades acima, é desejável a seguinte propriedade.

d) *Progresso:* Se algum valor foi proposto, então em algum momento um *learner* aprenderá algum valor.

Assim, temos que a propriedade (a) descarta soluções triviais, e.g., todos os processos aprendem valores pré-definidos. A propriedade (b) implica que valores aprendidos não podem ser desaprendidos. A propriedade de consistência (c) garante que o mesmo valor é aprendido por todos os processos. Quanto à propriedade (d) sabe-se que é impossível garanti-la em sistemas assíncronos na presença de falhas (FISCHER; LYNCH; PATERSON, 1985); para ser satisfeita, o sistema deve ser aumentado de alguma forma, por exemplo com sincronismo parcial (CHANDRA; HADZILACOS; TOUEG, 1996; DWORK; LYNCH; STOCKMEYER, 1988).

2.3 Paxos

Paxos é um protocolo de consenso bem conhecido, proposto por Leslie Lamport em 1998 (LAMPORT, 1998), e usado em diversos sistemas na indústria, e.g., *Chubby Lock Service* (BURROWS, 2006) e Doozer (MIZERANY; RARICK, 2011).

No *Paxos*, as propostas (comandos) são enviadas pelos clientes para qualquer *proposer*, porém apenas o Líder, um tipo especial de *proposer*, possui permissão de propor algum comando aos *acceptors* e consequentemente ter suas propostas decididas. Assim, o Líder escolhe uma das propostas para que futuramente se torne a decisão e, desde que não haja mais de um processo que se julgue Líder e que exista um *quorum* de *acceptors* corretos e alcançáveis, a instância sucederá.

O *Paxos* é executado em rodadas, e cada rodada possui duas fases, a fase de preparação (i.e. *phase one* ou *Phase1*) e a fase de aceite (i.e. *phase two* ou *Phase2*), como mostrado na Figura 1. Aqui descreveremos o algoritmo clássico do *Paxos*, comumente utilizado na implementação de replicação de máquinas de estados, também conhecido como *Multi-Paxos* (CHANDRA; GRIESEMER; REDSTONE, 2007). Nessa implementação o valor a ser acordado entre as réplicas, ou seja, entre os *acceptors*, é a ordem total das requisições enviadas pelos clientes. Tal ordem é estabelecida executando infinitas instâncias de *Paxos* e atribuindo um número de sequência a cada uma.

Uma instância, significa uma execução do algoritmo *Paxos* que pode vir a ter infinitas rodadas (uma nova rodada é criada apenas no caso de falha em se alcançar consenso na rodada anterior), e que futuramente tem um valor decidido, sendo que apenas um *proposer* em cada uma dessas instâncias atua como Líder, e portanto, apenas ele pode propor um valor. Caso o Líder venha a falhar, outro agente pode ser eleito e uma nova rodada é criada para essa instância.

A fase de preparação é iniciada pelo Líder enviando uma mensagem $Msg1A(r)$ para todos os *acceptors*, onde r é a rodada em que o Líder deseja propor algum valor. Quando um *acceptor* recebe uma mensagem $Msg1A(r)$ na *Phase1* ele pode realizar uma das seguintes ações:

- Se o *acceptor* ainda não respondeu a nenhuma mensagem $Msg1A(r)$, ele atualiza sua rodada para r e envia uma mensagem de confirmação ($Msg1B(r)$) para o Líder, assegurando assim, que posteriormente não aceitará valores para rodadas menores que r.

- Se o *acceptor* já respondeu à alguma mensagem $Msg1A(s)$ em uma rodada $s < r$, então dois cenários são possíveis:

  - O *acceptor* ainda não recebeu nenhuma mensagem $Msg2A$ com uma proposta, enviada durante a fase de aceite pelo Líder. Neste caso, o *acceptor* atualiza sua rodada para a maior recebida (i.e. r) e envia $Msg1B(r)$ para o Líder;

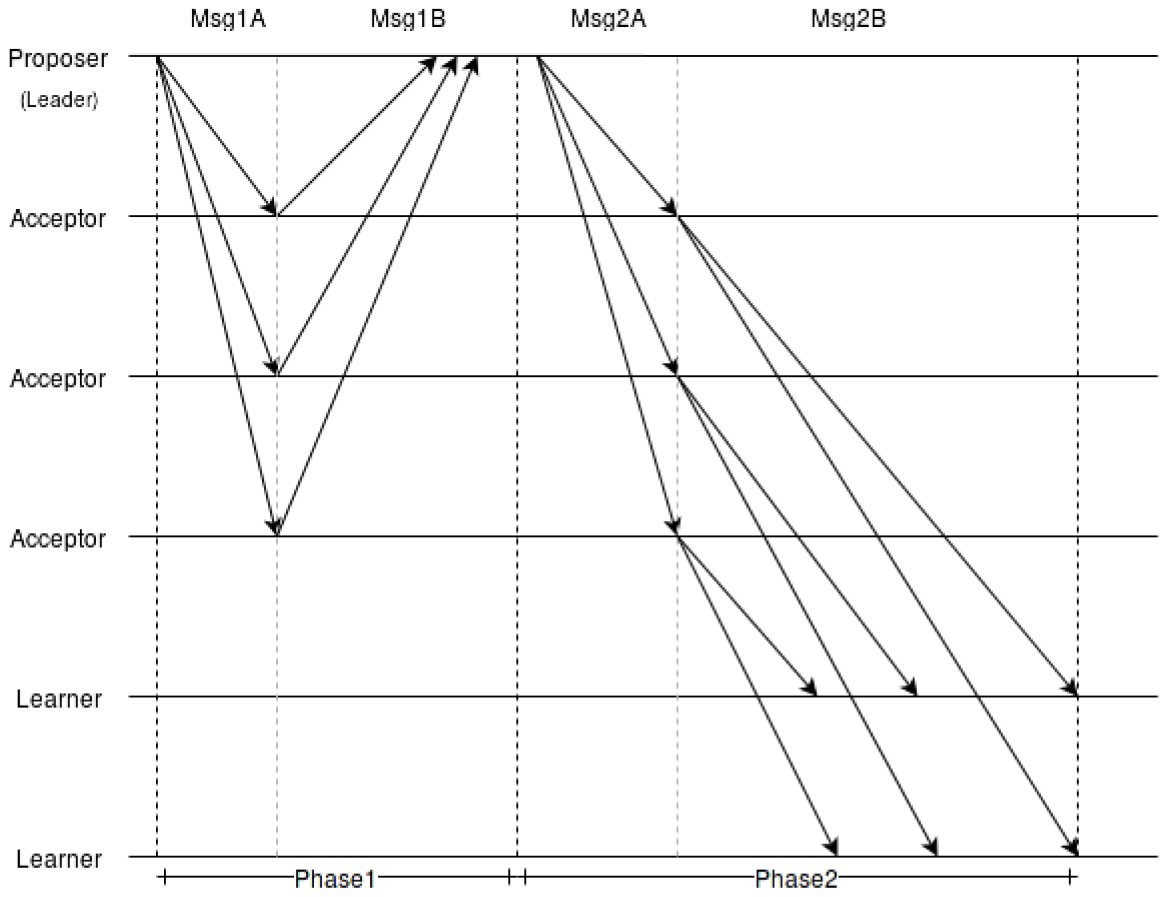


Figura 1 – Fluxo de mensagens do protocolo *Paxos* na ausência de falhas.

– O *acceptor* recebeu alguma mensagem $Msg2A(k)$ em uma rodada k e deve ter recebido um valor v proposto por algum *proposer* (que atue como Líder) nessa rodada ($k \leq s < r$), ou seja, v é o voto desse *acceptor* na última rodada k na qual ele votou. Assim, a proposta e a rodada recebida são incluídos na mensagem de resposta $Msg1B(r, k, v)$ ao Líder da rodada r.

A fase de aceite (*Phase2*) inicia após o Líder receber respostas de um *quorum* de *acceptors* durante a *Phase1*. De posse dessas mensagens o Líder pode determinar qual valor irá incluir na mensagem $Msg2A$ para ser proposto, usando os seguintes critérios:

- ❏ Se o Líder recebeu uma ou mais mensagens $Msg1B$ contendo valores já aceitos pelos *acceptors*, então ele seleciona o valor v da proposta que possuí a maior rodada (i.e. maior *proposal number*).
- ❏ Se nenhuma das mensagens $Msg1B$ recebida pelo Líder possuem algum valor aceito, o Líder pode escolher qualquer valor para propor.

Após selecionar um valor, o Líder realiza um *multicast* ou *broadcast* de mensagens $Msg2A(r, v)$ para um conjunto de *acceptors*. Quando um *acceptor* recebe uma mensagem $Msg2A(r, v)$, este a aceita como uma proposta somente se respondeu a priori à uma

mensagem $Msg1A(r)$. Então, o *acceptor* envia uma mensagem $Msg2B(r)$ confirmando o valor v aceito ao conjunto de *learners*.

Note que quando um *acceptor* aceita uma proposta, isto não significa que o valor contido nessa proposta foi decidido ou aprendido. Apenas após um *quorum* (e.g. uma maioria) de *acceptors* aceitar o mesmo valor proposto, é que um valor pode ser considerado decidido e será aprendido, em algum momento, por algum *learner* correto.

O protocolo apenas executa a *Phase1* novamente em caso de suspeita de alguma falha na rodada atual, o que exige que o Líder crie uma nova rodada. Além disso, esta fase pode ser executada em paralelo e *a priori* para múltiplas instâncias. Assim, durante a operação normal do protocolo, clientes enviam requisições para os *proposers* que as repassam para o Líder, que executa a *Phase2* do protocolo, futuramente obtendo um valor decidido ao final de cada instância e por fim, respondendo aos clientes. Logo, apenas a *Phase2* apresenta reais custos de passos de comunicação no protocolo. Uma proposta pode ser decidida em três passos de comunicação, se originada a partir de um *proposer* qualquer, ou em dois passos de comunicação, caso proposta diretamente pelo Líder.

Na Figura 2 é mostrada uma linha de execução do algoritmo *Paxos* na resolução de uma instância do problema de Consenso. P_x, na figura, representa um *proposer* cujo identificador único é x (um número Natural), o mesmo vale para os *Learners* (L_x) que estão associados a uma aplicação final, como uma base de dados por exemplo. Os *acceptors* são representados pela letra A, e mostrados como um aglomerado, as informações sobre o *quorum* foram omitidas por brevidade. C representa o Líder dessa rodada para essa instância do protocolo. Os losangos coloridos são valores (ou comandos) propostos por clientes aos *proposers*.

O fluxo inicia com o recebimento de algum comando por algum *proposer* advindo de algum cliente a ele conectado. No exemplo, dois comandos são recebidos, o losango verde e o amarelo. Ao receber um comando, o *proposer* encaminha esse comando para o Líder (que pode ser uma mensagem interna, pois ser Líder é apenas uma função extra desempenhada por algum *proposer* que acredite ser o Líder da rodada).

Como dito anteriormente, no *Paxos*, apenas o Líder pode propor algum valor, no exemplo, ao receber os dois valores o Líder deve escolher um, e pode propô-lo na primeira instância na qual ele ainda não propôs ou aprendeu algo. Supondo que o *Paxos* executou normalmente, sem a ocorrência de falhas, em algum momento algum valor, no caso o losango amarelo, será aprendido por todos os *learners* não falhos, em dois passos de comunicação se o Líder for P_0 (a mensagem de P_0 para C seria uma ação interna) e em três passos de comunicação se P_1 for o Líder (P_0 encaminha sua proposta para C, que na verdade é P_1).

Um problema com essa abordagem é que um comando que havia sido recebido (losango verde) na mesma instância não é aprendido devido a colisão. Sendo necessário que tal comando seja reproposto pelo Líder em uma próxima instância, causando assim um maior

tempo de espera pela decisão.

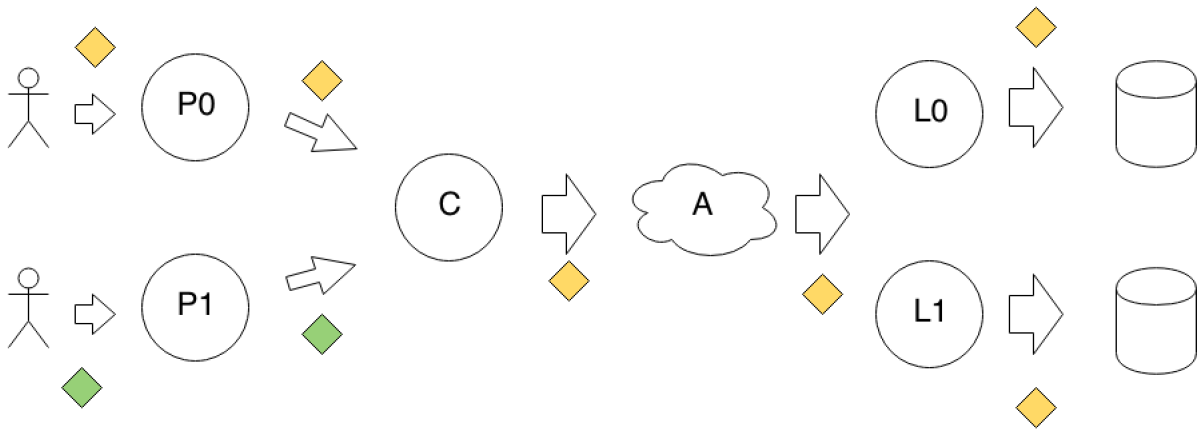


Figura 2 – Exemplo de execução da *Phase2* de uma instância do protocolo *Paxos*.

2.4 Fast Paxos

Algoritmos que contornam o Líder com o intuito de conseguir uma latência de dois passos de comunicação para múltiplos *proposers* (na *Phase2*), são conhecidos como protocolos rápidos (*fast*). Porém, contornar o Líder pode resultar em colisões, isto é, propostas concorrentes que possivelmente impedem que qualquer decisão seja alcançada após dois passos de comunicação. Além disso, resolver a colisão exige o uso de *quorums* maiores, tal que cada três tenham uma intersecção não vazia (LAMPORT, 2006a).

O objetivo do *Fast Paxos* (LAMPORT, 2006a) é justamente reduzir essa latência fim-a-fim para se alcançar *consensus*, possibilitando que os demais *proposers*, e não apenas o Líder, proponham valores a serem aceitos pelos *acceptors* e aprendidos pelos *learners* em dois passos.

Assim como o *Paxos*, o *Fast Paxos* opera em rodadas e cada rodada tem duas fases, sendo que, como mostrado em (LAMPORT, 1998), a *Phase1* pode ser executada uma única vez para todas as infinitas instâncias do algoritmo desde que inicialmente exista apenas um Líder. Portanto, o custo de se alcançar um consenso está na segunda fase do algoritmo (i.e. *Phase2*). E é justamente nessa fase que vários esforços foram gastos na tentativa de reduzir a latência de se alcançar um acordo.

No *Fast Paxos* há dois tipos de rodadas: a clássica e a rápida. A Figura 3 ilustra os passos de comunicação de cada uma dessas rodadas para a *Phase2* do protocolo.

Uma rodada clássica ocorre como no *Paxos*: um *proposer* propõe um valor para o Líder (o que pode ser uma mensagem interna caso este *proposer* seja o Líder), que por sua vez o repassa para um conjunto de *acceptors*, estes, podem vir a aceitar o valor (eles podem não aceitar valor algum caso existam multiplos Líderes). Cada *acceptor* aceita algum valor e o repassa como um "voto" para os *learners*, que por sua vez, ao receberem

o mesmo valor de um *quorum* de *acceptors* (quantidade de votos necessárias) aprendem o valor proposto, chegando a um consenso em um total de três passos de comunicação.

O fluxo em azul na Figura 3 ilustra uma rodada clássica. Já uma rodada rápida, é similar a uma clássica, exceto pelo fato que um passo de comunicação a menos é necessário para se atingir um acordo, isso porque, em uma rodada rápida, os *proposers* não precisam propor um valor para o Líder, podem propor diretamente para o conjunto de *acceptors*, como mostrado no fluxo em rosa na Figura 3, aprendendo o valor em apenas dois passos de comunicação na ausência de falhas.

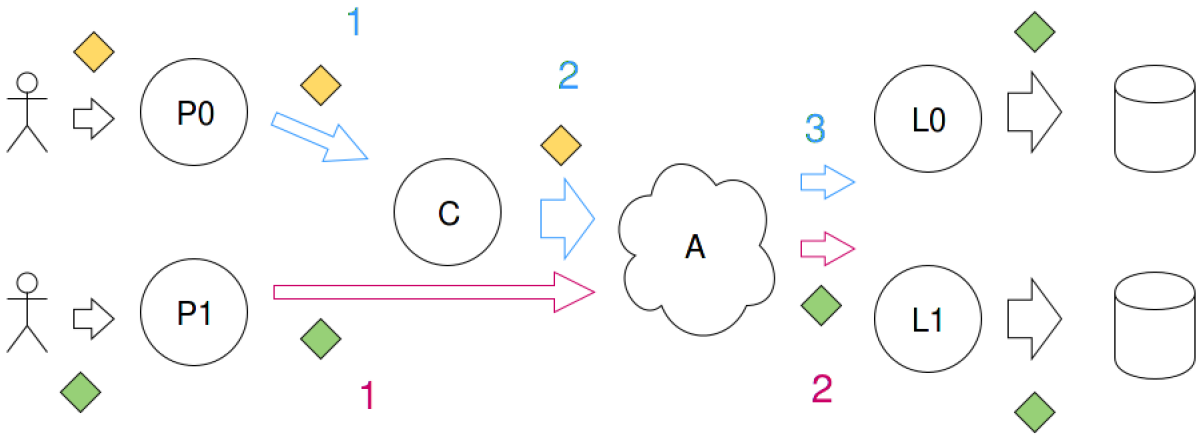


Figura 3 – Exemplo de execução da *Phase2* de uma instância do protocolo *Fast-Paxos*.

Ao permitir que os *acceptors* recebam as propostas diretamente de um *proposer* (provida por um cliente) e não exclusivamente pelo Líder, uma rodada rápida economiza um passo de comunicação. Como uma réplica pode desempenhar o papel de *proposer* e *learner* ao mesmo tempo, tal abordagem pode reduzir a latência fim-a-fim para o aprendizado.

Porém como o *Paxos* é provadamente ótimo (LAMPORT, 2001), para se reduzir a latência no *Fast paxos* é preciso sacrificar a resiliência, sendo necessários um total de $3F + 1$ *acceptors* para se tolerar F *acceptors* falhos e garantir consistência. Assim, os *learners* precisam receber o aceite para um mesmo valor de pelo menos $2F + 1$ *acceptors* para aprender esse valor, enquanto que no *Paxos*, apenas $F + 1$ *acceptors* são necessários na formação de um *quorum*.

Além disso, como um *proposer* qualquer pode enviar sua proposta diretamente para os *acceptors*, diferentes *acceptors* podem vir a aceitar diferentes valores. Este cenário é conhecido como uma colisão (LAMPORT, 2006a) e pode levar a nenhum valor decidido ao final da instância.

Evitar colisões e se recuperar delas são novos problemas que surgiram com o advento de protocolos *fast*, como o *Fast Paxos*. Em (LAMPORT, 2006b) são apresentadas duas condições nas quais protocolos assíncronos de Consenso podem decidir em dois passos de comunicação, mesmo em presença de colisões, juntamente com protocolos *Collision-Fast* que exploram tais condições. Porém tais protocolos impõem restrições e continuam

tendo o problema de que apenas um valor dentre os propostos pode ser decidido em cada instância de Consenso. Assim, se múltiplos valores precisam ser decididos, como no caso da implementação de difusão atômica, valores propostos e não decididos, devem ser repropostos em instâncias subsequentes (SCHMIDT; CAMARGOS; PEDONE, 2007).

2.5 Difusão Atômica

O problema de difusão atômica consiste em garantir que todas as mensagens enviadas a um conjunto de processos são entregues em uma mesma ordem total. Formalmente, o problema também pode ser definido em termos dos mesmos agentes e papéis que o consenso. Assim o problema de difusão atômica pode ser entendido como o acordo em uma sequência crescente de mensagens difundidas por *proposers*, em que *learners* aprendem incrementando prefixos nessa sequência.

Uma sequência s de tamanho n é representada por $\langle v_1, v_2, \cdots, v_n \rangle$, onde v_i é o i-ézimo elemento da sequência. Uma sequência s é dita prefixo de uma sequência t, denotado por $s \sqsubseteq t$ se e somente se o tamanho de s é menor ou igual ao tamanho de t e, para todo i de 1 até o tamanho de s, $s[i] = t[i]$; s e t são iguais, se e somente se, $s \sqsubseteq t$ e $t \sqsubseteq s$. A sequência vazia $\langle\rangle$ tem tamanho zero e é prefixo de todas sequências.

O problema de difusão atômica pode então ser definido pelas seguintes propriedades, onde *delivered[l]* se refere à sequência de mensagens entregues pelo *learner* l, inicialmente $\langle\rangle$.

a) *Não trivialidade:* Para qualquer *learner* l, *delivered[l]* contém apenas mensagens difundidas e não duplicadas.

b) *Estabilidade:* Para qualquer *learner* l, se *delivered[l]*$= s$ em algum momento, então $s \sqsubseteq$ *delivered[l]* em todos momentos posteriores.

c) *Consistência:* Para qualquer par de *learners* $l1$ e $l2$, ou *delivered[*$l1$*]* $\sqsubseteq$ *delivered[*$l2$*]* ou *delivered[*$l2$*]* $\sqsubseteq$ *delivered[*$l1$*]*.

O progresso do algoritmo pode ser definido em termos de outro conjunto de agentes: os *acceptors*. Seja um *quorum* qualquer conjunto finito de *acceptors* suficientemente grande de forma a garantir o progresso em um sistema parcialmente síncrono. Tem-se então que:

d) *Progresso:* Para qualquer *proposer* p e *learner* l, se p, l e um *quorum* de *acceptors* são corretos e p difunde uma mensagem m, então em algum momento *delivered[l]* contém m.

Difusão Atômica pode ser solucionado por meio de uma ordenação total de infinitas instâncias de Consenso (CHANDRA; TOUEG, 1995). Em cada instância, um ou mais

proposers podem propor um valor que futuramente será aprendido por todos os *learners* corretos. Um *broadcast* é realizado por um *proposer* propondo um valor na primeira instância de Consenso em que este ainda não propôs ou aprendeu algo. Para que o *proposer* possa aprender, ou seja, saber que um valor foi decidido, ele deve ser também um *learner*. Assim todos os *learners* em algum momento aprenderão uma sequência de valores acordados, em uma mesma ordem total.

Cada uma das infinitas instâncias de *consensus* pode ser resolvida por uma execução de *Paxos*, e a solução de cada execução do *Paxos* forma uma sequência totalmente ordenada no nível do protocolo de *Atomic Broadcast*, como mostrado na Figura 4. Nela, os seis comandos concorrentes, propostos por três clientes, foram aprendidos ao final de 6 instâncias de *consensus*, uma para cada comando proposto, por todos os *learners* não falhos. Essa execução levou seis instâncias para finalizar devido ao fato de apenas o Líder poder propor algum comando, obrigando-o a re-propor valores não decididos, desta forma, o Líder atua como centralizador de propostas.

No exemplo, a ordem de chegada das mensagens é representada pela posição dos losangos na figura, ou seja, os comandos representados pelos losangos azul e verde são concorrentes, assim como os comandos laranja, amarelo e rosa, e portanto apenas um desses comandos pode ser proposto por C e em algum momento aprendido por instância de Consenso, os demais devem ser repropostos em uma nova instância. Já o comando roxo, não concorre com nenhum dos demais. Cada quadrado cinza representa uma instância de *consensus* resolvida e já aprendida pelos *learners*, e o losango inscrito representa o valor decidido naquela instância.

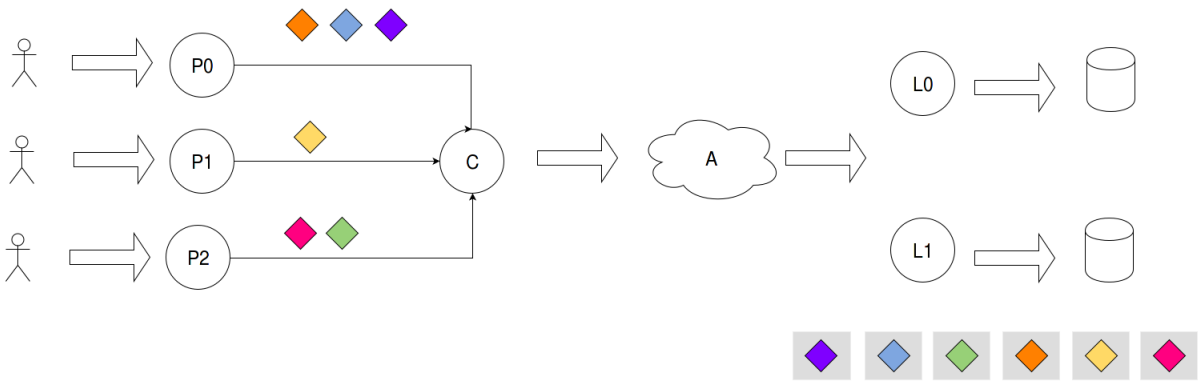


Figura 4 – Devido a ocorrência de re-proposições pelo Líder são necessárias seis instâncias de *consensus* para que todos os valores propostos sejam aprendidos, considerando uma implementação de *Atomic Broadcast* que utilize *Paxos* com um Líder.

2.6 M-Consensus

O problema de *M-Consensus*, onde M significa mapeamento (ou função de mapeamento), é uma extensão do problema de *Consensus* distribuído proposta em (SCHMIDT;

CAMARGOS; PEDONE, 2007), cujo acordo é realizado em um conjunto de valores (mapa de valores) e não sobre apenas um valor por instância. Essa característica permite sanar o problema de colisões existente nos protocolos *fast* citados na Seção 2.4, já que um algoritmo que resolve *M-Consensus* consegue decidir em dois passos de comunicação, e em mais de um valor por instância, sem a necessidade de re-proposições.

Ao se implementar um algoritmo de difusão atômica utilizando o protocolo *Collision-fast Paxos*, uma variante do *Paxos* que resolve *M-Consensus*, obtém-se uma solução que garante melhor desempenho, e mesma resiliência de um protocolo de difusão atômica que utiliza *Paxos*, sem a necessidade de um agente centralizador de propostas (i.e. Líder).

No *M-Consensus*, os agentes devem concordar em um crescente mapeamento de *proposers* para valores propostos ou *Nil*. Essa função de mapeamento é representada por uma estrutura de dados denominada *v-mapping* (i.e. mapa de valores). Cada *proposer* pertencente ao conjunto de *proposers* (conjunto Domínio) é mapeado para algum valor do conjunto de valores possíveis ou o valor especial *Nil* (conjunto Imagem) (SCHMIDT; CAMARGOS; PEDONE, 2007); tal relação é ilustrada na Figura 5.

Assim um *v-mapping*, como por exemplo: $vmap = \{(P_0, V_0), (P_1, Nil)\}$, é um subconjunto de todos os possíveis conjuntos de pares $\langle Proposers, Values \rangle$. Logo, existe um conjunto *ValMap* de *v-mappings*, em que cada *v-mapping* é definido como todas as funções que mapeiam um subconjunto de *proposers* para um valor ou *Nil*. Formalmente, esse conjunto pode ser definido como mostrado na Equação 1:

$$ValMap = \bigcup \{[P \mapsto V] : P \in Proposers \wedge V \in \{Values \cup \{Nil\}\}\} \tag{1}$$

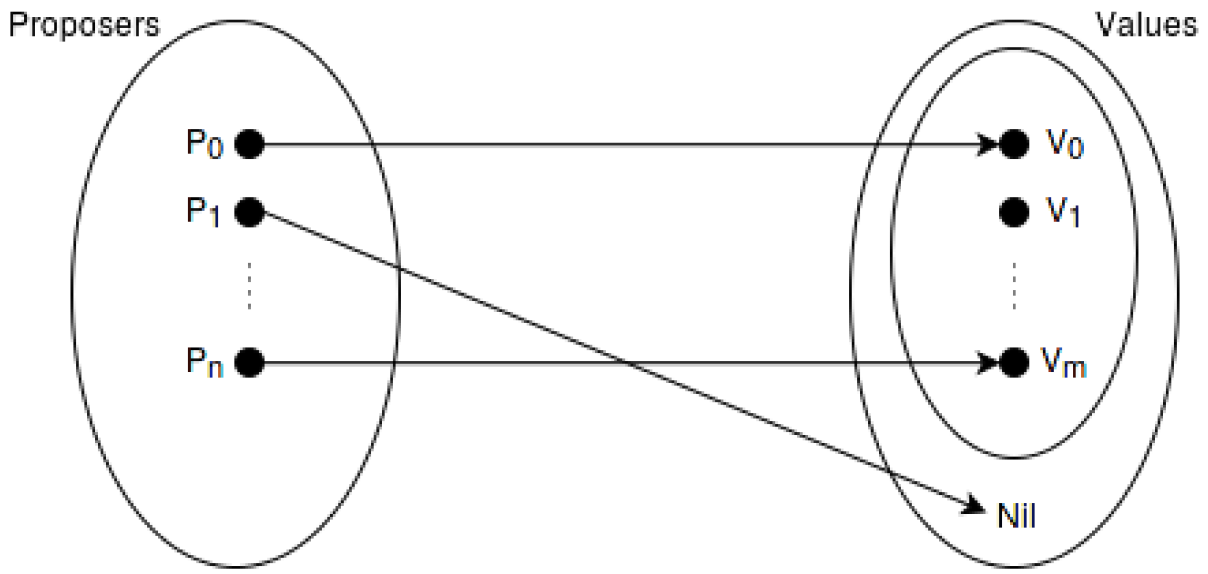


Figura 5 – ValMap: Conjunto de mapas de valores (*v-mappings*)

Os mapas possuem uma relação de precedência, de forma que é possível definir uma ordem entre eles (e.g. usando um identificador único para cada *proposer*). Caso um mapa

contenha apenas um elemento no seu Domínio, ou seja, mapeie apenas um *proposer* para um valor ou *Nil*, esse mapa é dito simples, ou *s-mapping*.

Neste contexto, *proposers* propõem valores e *learners* aprendem *v-mappings*, que podem ser diferentes, mas são sempre compatíveis, podem ser estendidos e em algum momento devem se tornar iguais.

Por compatível entende-se um *v-mapping* cujos elementos do Domínio mapeiam para os mesmos valores de outro *v-mapping*, ou seja, dados dois *v-mappings*, eles são compatíveis se os elementos na intersecção de seus Domínios mapeiam para os mesmos valores.

Como mostrado em (SCHMIDT; CAMARGOS; PEDONE, 2007), um *v-mapping* w pode ser estendido de forma a aumentar o seu Domínio utilizando uma operação de *append*, acrescentando à w um *s-mapping* c, denotado por $w \bullet \langle p, V \rangle$, tal que $p \in$ Proposers e $V \in \text{Values} \cup \{Nil\}$.

O operador *append* define uma relação de ordem parcial entre os *v-mappings*, de modo que dado dois *v-mappings* v e w, dizemos que v é prefixo de w (representado por: $v \sqsubseteq w$) se, e somente se, w pode ser obtido através de uma série de operações de *append* em v.

Um *v-mapping* é dito completo se, e somente se, todo *proposer* pertence ao seu Domínio. Consequentemente, um *v-mapping* completo não possui extensão, já que não é possível realizar uma operação de *append* que resulte em um *v-mapping* diferente.

Se um *v-mapping* completo mapeia todo elemento do Domínio para *Nil*, este é dito *Trivial* e é independente do conjunto de valores possíveis. Logo, um *v-mapping* é "Não-trivial" se, e somente se, é diferente de um *trivial*.

Temos então que as seguintes propriedades do problema de *M-Consensus* podem ser definidas, onde $learned[l]$ representa o *v-mapping* que foi aprendido por um *learner* l, inicialmente vazio:

a) *Não trivialidade:* Para qualquer *learner* l, $learned[l]$ é sempre um *v-mapping* proposto e "não-trivial".

b) *Estabilidade:* Para qualquer *learner* l, se $learned[l] = v$ em um dado momento, então v é prefixo de $learned[l]$ em todos os momentos posteriores.

c) *Consistência:* O conjunto de *v-mappings* aprendido é sempre compatível e "não-trivial".

O progresso no *M-Consensus* pode ser definido pela seguinte propriedade:

d) *Progresso:* Para qualquer *proposer* p e *learner* l, se p, l e um *quorum* de *acceptors* não são falhos e p propõe um valor, então $learned[l]$ se tornará completo em algum momento.

É possível notar que ambos os problemas são equivalentes. Um algoritmo que resolve *Consensus* pode resolver *M-Consensus* tendo os *learners* aprendendo um mapa em que

um *proposer* é mapeado para o valor decidido e todos os demais *proposers* são mapeados para *Nil*. Assim como um algoritmo que resolve *M-Consensus* trivialmente resolve *Consensus*, ordenando totalmente o conjunto de *collision-fast proposers* e escolhendo o valor do primeiro *collision-fast proposer*, diferente de Nil, como decisão.

Desta forma, a vantagem do *M-Consensus* sobre o *Consensus* clássico, é que este permite que duas ou mais propostas concorrentes sejam decididas em uma mesma instância, mapeadas para diferentes *proposers*, evitando assim colisões.

2.7 Collision-fast Paxos

O algoritmo *Collision-fast Paxos*, proposto em (SCHMIDT; CAMARGOS; PEDONE, 2007) e implementado nesse trabalho, resolve o problema de colisões solucionando uma instância de *M-Consensus* e, na ausência de falhas, mantêm uma latência ótima de dois passos de comunicação, sendo portando, um protocolo *collision-fast*.

2.7.1 Algoritmo básico

As rodadas são totalmente ordenadas por uma relação de precedência $\leq$ e contém informação sobre quem é o coordenador e qual é o conjunto de *Collision-fast Proposers* dessa rodada.

O conjunto de *Collision-fast proposers* é um subconjunto do conjunto de *proposers* de uma rodada que são os únicos *proposers* que podem ter suas propostas decididas em dois passos de comunicação (denominada proposta rápida). Um *Collision-fast proposer* p pode propor de forma rápida (*fast-propose*) um *s-mapping* $\langle p, V \rangle$ quando:

- possui algum valor V para ser proposto, tal que $V \neq Nil$ (e.g. recebeu algum comando de *proposer* comum, e.g., cliente), ou
- percebe que algum outro *Collision-fast proposer* em uma mesma rodada propôs de forma rápida um valor diferente de *Nil* e, neste caso, p propõe de forma rápida $\langle p, Nil \rangle$.

Se essa proposta rápida contém um mapa com um valor V diferente de *Nil*, então o *Collision-fast proposer* envia V para o conjunto de *acceptors* e demais *collision-fast proposers* da rodada, caso contrário, ele envia o valor *Nil* diretamente para os *learners*.

Um *acceptor* pode aceitar múltiplos mapas de valores (*v-mappings*) desde que os novos mapas estendam os previamente aceitos, por meio de operações de *append*. Logo, um *v-mapping* v é prefixo de um *v-mapping* w, e consequentemente, w estende v ($v \sqsubseteq w$) se, e somente se, w pode ser construído a partir de consecutivas operações de *append* em v.

Os *v-mappings* aceitos por um *acceptor* são construídos a partir de um *s-mapping não-trivial* que tenha sido proposto de forma rápida e, portanto, sempre possuem um

mapa que mapeia pelo menos um *proposer* para um valor diferente de *Nil*, garantindo a não trivialidade da decisão.

Assim, um *v-mapping* é dito aceito em uma rodada r se ele é prefixo de todo *v-mapping* aceito por um *quorum* de *acceptors* em r. Portanto, tal *v-mapping* é garantidamente compatível e um *learner* pode seguramente estender *learned[l]*, aprendendo tal *v-mapping*.

Se pelo menos um *Collision-fast proposer* propõe de forma rápida um valor, nenhum agente é falho e as mensagens são entregues corretamente, é possível notar que os *learners* aprendem um *v-mapping* não trivial em dois passos de comunicação.

É importante notar também, que se um *v-mapping* foi aceito em uma determinada rodada, é sempre possível estendê-lo em rodadas posteriores. Isto é garantido pelas ações tomadas pelo coordenador durante o inicio de uma nova rodada.

Para criar uma nova rodada, um coordenador consulta um *quorum* de *acceptors* para verificar se algum *v-mappings* foi ou pode ser aceito em alguma rodada anterior. Se for esse o caso, o coordenador pode estender tal *v-mapping* com valores *Nil* para torná-lo completo e o enviar para os *acceptors* para serem aceitos e posteriormente aprendidos. Caso nenhum *v-mapping* tenha sido aceito em rodadas anteriores, os *collision-fast proposers* da rodada atual são informados que podem propor de forma rápida nessa rodada.

Para que o coordenador seja capaz de identificar que algum valor foi ou pode ser escolhido em uma rodada menor que a atual apenas consultando um *quorum* de *acceptors*, a seguinte hipótese se faz necessária:

Assumption 1 ***(Exigência do* quorum*)****:** Se Q e R são* quorums*, então $Q \cap R \neq \emptyset$*

De fato, qualquer algoritmo que se proponha a resolver *Consensus* assíncrono deve satisfazer uma condição similar a Exigência de *quorum* 1, como mostrado em (LAMPORT, 2006b). Uma forma simples de se garantir tal exigência é definindo um *quorum* como qualquer maioria de *acceptors*.

O Algoritmo 2.1, proposto em (SCHMIDT; CAMARGOS; PEDONE, 2014), apresenta o *Collision-fast Paxos* em detalhes, onde $a \bullet S$ denota uma aplicação cumulativa de *appends* nos elementos do conjunto S (em qualquer ordem). Por exemplo, $a \bullet \{x, y\} = a \bullet x \bullet y$.

O algoritmo é apresentado como uma sequencia de ações, realizadas apenas se todas as *pré-condições* são satisfeitas. O fluxos de ações é executado em rodadas, sendo que na ausência de falhas, uma única rodada é necessária para se alcançar um consenso.

Em cada rodada há duas fases, similar ao descrito na Seção 2.3. Na *Phase1*, usada para (re)configuração do protocolo, o Líder consulta o conjunto de *acceptors* para saber qual valor foi ou pode ser aceito, porém, no caso do *Collision-fast paxos*, o valor é um *v-mapping*, ou seja, um conjunto de valores. Essa ação é descrita na linha 6 do Algoritmo 2.1 no método *Phase1a* e a resposta dos *acceptors* é tratada na linha 14 pelo método *Phase1b*.

Após essa fase de preparação, executada apenas no início do protocolo ou quando ocorre uma falha no Líder, a *Phase2* pode ser iniciada. O Líder então executa a *Phase2Start*

preparando os *acceptors* para novas propostas, normalizando entre os participantes os valores previamente decididos, e informando aos *proposers* (*Phase2Prepare*) que podem propor nessa rodada.

Uma proposta é disparada pelo método *Propose*, por algum *proposer* que queira propor algum valor (recebido de algum cliente). Se esse for o caso, ele repassa essa proposta para um *Collision-fast proposer* (essa mensagem pode ser interna, caso esse *proposer* seja *collision-fast*). Assim, os *Collision-fast proposers* executam o método *Phase2a*, propondo o valor na primeira instância em que ele ainda não propôs ou aprendeu algo. Essa proposta é tratada pelos *acceptors* no método *Phase2b*, em que os *acceptors* verificam se podem aceitar tal proposta e, caso afirmativo, estendem o que já foi decidido e repassam esse novo *v-mapping* para os *learners*.

Se tudo acontece no seu fluxo normal, na ausência de falhas, em algum momento *v-mappings* serão aprendidos (método *Learn*) por todos os *learners*, estendendo o *v-mapping* já aprendido com o maior prefixo comum recebido de um *quorum* de *acceptors*, mantendo o aprendizado compatível, e posteriormente, entregando a decisão para os clientes.

Caso alguma falha ocorra, e seja reportada pelo detector de falhas, o coordenador pode iniciar uma nova rodada. Porém, tal método de recuperação de falhas pode aumentar o número de passos de comunicação para se decidir um valor, já que exige a execução da *Phase1* do protocolo novamente.

Além disso, tal procedimento não garante que a nova rodada sucederá, já que o(s) agente(s) falho(s) pode(m) ainda estar(em) presente(s) na nova rodada. Como o conjunto de *Collision-fast proposers* pode ser escolhido a cada rodada pelo coordenador, e tal conjunto é necessário para que o protocolo seja *collison-fast*, o coordenador pode excluir da nova rodada os *proposers* suspeitos de falhas, permitindo que o protocolo volte a ser *Collision-fast* rapidamente. Assim, o protocolo consegue se recuperar de falhas e se estabilizar rapidamente, permitindo que a latência em se atingir consenso volte a ser ótima (dois passos de comunicação).

Ademais, *acceptors* e *learners* suspeitos também podem ser excluídos de rodadas futuras, porém o protocolo não especifica essas ações. Mas esse trabalho implementa tais ações (detalhadas na Seção 3.1.2.1) em que o módulo *membership manager* com o auxilio do detector de falhas consegue prever se um agente está "saudável" e gerenciar o conjunto de agentes.

A Figura 6 ilustra um fluxo de execução da *Phase2* do protocolo *Collision-fast Paxos* no qual três *proposers* desempenham o papel de *collision-fast proposer* para a rodada em questão, mas apenas dois deles (P_0 e P_2) receberam uma requisição de algum cliente.

No *Collision-fast Paxos* não é o Líder quem propõe valor aos *acceptors*; tal função é designada exclusivamente aos *Collision-fast proposers*, possibilitando que propostas concorrentes sejam aprendidas, sem ocasionar colisões. Como mais de um valor (losango) pode ser aprendido por instância, sendo, cada valor mapeado para um *Collision-fast*

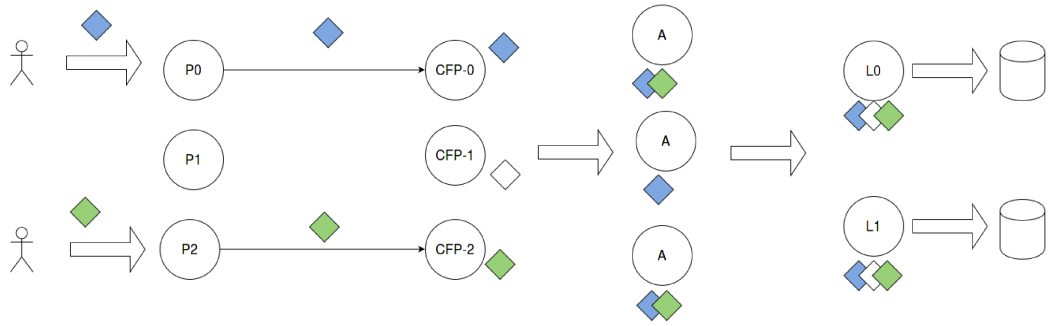


Figura 6 – Exemplo de execução da *Phase2* de uma instância do protocolo *Collision-fast Paxos*.

proposers distinto, um *v-mapping* pode ser aprendido em dois passos de comunicação, contendo valores de multiplos *proposers*.

Os valores representados pelos losangos verde e azul foram propostos de forma concorrente pelos *Collision-fast proposers* CFP_0 e CFP_2. Tais valores foram aceitos pelos *acceptors*, que estenderam seu conjunto de valores aceitos para essa instância, e em algum instante, os *learners* L_0 e L_1 receberam mensagens Msg2B com tais *v-mappings*, suficientes para formar um *quorum*, aprendendo esse conjunto de valores que pode ser expresso como: $\{P_0 \rightarrow azul, P_1 \rightarrow Nil, P_2 \rightarrow verde\}$.

Note que P_1 não possuía nenhum valor para propor nessa instância (decorrente de alguma requisição cliente), e portanto, por ser um *collision-fast proposer* dessa rodada, deve enviar um valor especial *Nil* diretamente para os *learners* e demais *collision-fast proposers*, informando que ele se abstêm de propor algum *valor* $\neq$ *Nil* nessa instância.

Tal procedimento é necessário para garantir o progresso do protocolo, permitindo que todo *learner*, em algum momento, aprenda um *v-mapping* completo. Caso P_1 não fosse um *collision-fast proposer* dessa rodada, ele não precisaria enviar *Nil* para os *collision-fast proposers* e *leaners*, pois os *learners* já saberiam que P_1 não pode propor (já que não é um *Collision-fast proposer*), e neste caso, atribuem *Nil* a P_1 automaticamente, como mostrado na linha 78 do Algoritmo 2.1 no método *Learn*.

Portanto, é possível notar que esse protocolo consegue atingir uma decisão em presença de colisões, com a mesma latência ótima de dois passos de comunicação que o *Fast-paxos*, e que não necessita de re-proposições. Isso possibilita ao *Collision-fast Paxos* manter uma latência ótima mesmo com o aumento do número de requisições concorrentes.

Além disso, como pode ser visto na Tabela 1, o *Collision-fast Paxos* atinge tal objetivo usando um modelo de tolerância a falhas igual ao do *Paxos* e mais fraco que o *Fast-paxos*, i.e., sem a necessidade de *quorums* maiores. Não obstante, o *Collision-fast Paxos* permite que mais de um valor seja aprendido por instância, aumentando a vazão com relação aos demais protocolos, pois resolve uma variante do *Consensus*, o *M-consensus*.

	Paxos (clássico)	Fast Paxos	Collision-fast Paxos
Passos de comunicação	3	2	2
Número de acceptors	2F + 1	3F + 1	2F + 1
Tamanho do *quorum*	F + 1	2F + 1	F + 1

Tabela 1 – Para cada protocolo: Número de passos de comunicação necessários para se alcançar acordo; número de réplicas desempenhando o papel de *acceptors* necessárias para garantir corretude; e tamanho mínimo de um *quorum* para se garantir progresso. Assumindo que o Líder não é alterado e que são toleradas no máximo F *acceptors* falhos.

2.8 Collision-Fast Atomic Broadcast

Como descrito na Seção 2.5, protocolos de *Atomic Broadcast* são comumente implementados utilizando *Paxos*, o que pode comprometer o desempenho do protocolo e consequentemente da aplicação que o utiliza para replicação, devido aos problemas previamente citados neste trabalho.

O algoritmo de Difusão Atômica apresentado em (SCHMIDT; CAMARGOS; PEDONE, 2007; SCHMIDT; CAMARGOS; PEDONE, 2014) é *Collision-Fast*. Esse algoritmo é baseado na variação do problema de *Consensus*, *M-Consensus*, apresentado na Seção 2.6. No qual processos devem decidir em um conjunto de valores propostos e não em um único valor.

A solução para tal problema, chamada *Collision-Fast Paxos*, apresentada na Seção 2.7, estende o protocolo *Paxos*, tolerando o mesmo número de falhas que o original e permitindo que múltiplos processos, e não apenas o Líder, possam ter suas propostas aprendidas em dois passos de comunicação.

O *Collision-Fast Paxos* pode ser usado na implementação de um protocolo de Difusão Atômica utilizando sucessivas instâncias de *M-Consensus*, assim como é feito na redução convencional do problema de Difusão Atômica para Consenso. Como múltiplas mensagens podem fazer parte de cada decisão do *Collision-Fast Paxos*, múltiplas mensagens podem ser entregues após cada instância, em dois passos de comunicação pelo *Collision-fast Atomic Broadcast* (SCHMIDT; CAMARGOS; PEDONE, 2007).

Assim, nesse trabalho, ao invés de se solucionar infinitas instâncias de *Consensus* a implementação do protocolo de *Atomic Broadcast* resolve infinitas instâncias da variante, o *M-Consensus*.

Como mostrado no Algoritmo 2.2, presente em (SCHMIDT; CAMARGOS; PEDONE, 2014), cada instância é unicamente identificada por um número natural i e a solução de cada uma dessas instâncias forma uma sequência de comandos de *Broadcast*. O agrupamento determinista de tais soluções compõe uma sequência de comandos totalmente ordenada na camada do protocolo de *Atomic Broadcast* da nossa implementação.

Como descrito na Seção 2.7, o conjunto de *Collision-fast proposers* é escolhido a cada

nova rodada pelo coordenador, permitindo que este exclua *proposers* suspeitos de falhas de novas rodadas. Assim, para se garantir progresso, assume-se que existe um coordenador c que acredita ser o Líder e que mantêm um conjunto *activep[c]* que contém todos os *proposers* que ele acredita serem alcançáveis.

O Algoritmo 2.2 pode garantir progresso nas mesmas situações que o *Paxos*, se para todo coordenador, ele for o único *Collision-fast proposer* das infinitas instâncias as quais ele coordena. Assim, é preciso que as seguintes condições sejam satisfeitas para obter-se tal garantia:

- ❏ $\{p, l, c\} \cup Q$ não são falhos, onde p significa algum *proposer*, l um *learner*, c um coordenador e Q um *quorum* de *acceptors*.
- ❏ p propôs um valor.
- ❏ c é o único coordenador que acredita ser o Líder (existe apenas um Líder).
- ❏ Todos os *proposers* em *activep[c]* são "saudáveis".
- ❏ *activep[c]* é um subconjunto de todos os possíveis conjuntos de *proposers*.

Logo, se todas as condições são satisfeitas para algum *proposer* p, coordenador c e *quorum* Q em algum momento, então em algum instante futuro o *learner* l aprende um *v-mapping* completo.

A Figura 7, ilustra o número de instâncias necessárias para se solucionar o mesmo cenário de concorrência apresentado na Figura 4 da Seção 2.5, porém utilizando o protocolo *Collision-fast Atomic Broadcast*, que consegue decidir sobre todos os valores propostos em menor tempo de execução, pois necessita de uma menor quantidade de instâncias para decidir sobre o mesmo conjunto de propostas, sem precisar efetuar re-proposições.

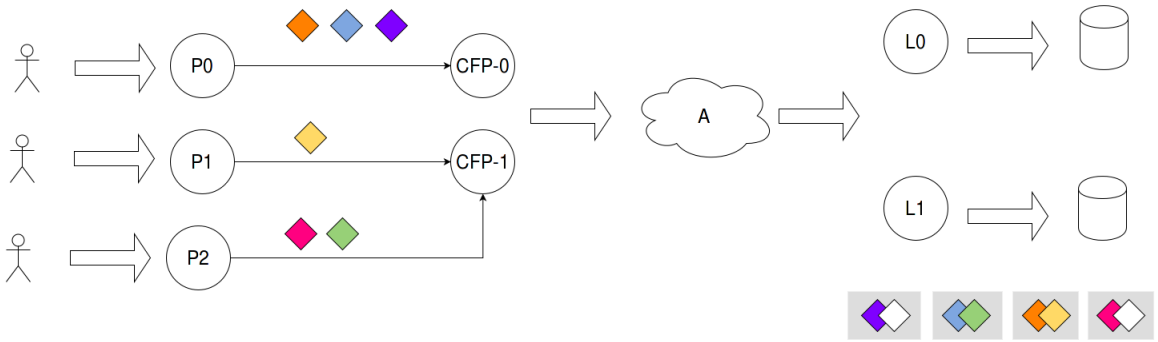


Figura 7 – Na execução do protocolo de *Atomic Broadcast* que utiliza *Collision-fast Paxos*, com dois *Collision-fast proposers*, foram necessárias quatro instâncias de *M-Consensus* para que todos os valores propostos fossem aprendidos. O número mínimo de instâncias, neste exemplo, é três.

Em um sistema de máquinas de estados replicadas implementado com base no protocolo de Difusão Atômica, clientes fazem o *broadcast* de comandos para réplicas que os

entregam e, de forma determinista, executam a sequência totalmente ordenada de comandos decididos, possibilitando que todas as réplicas cheguem a um mesmo estado. O desempenho do sistema replicado depende do desempenho do protocolo de Difusão Atômica, assim, um protocolo que garanta a entrega de comandos em menor número de passos de comunicação colabora para um melhor desempenho em tais sistemas. Dois outros protocolos existentes na literatura possuem semelhanças ao proposto em (SCHMIDT; CAMARGOS; PEDONE, 2007): Mencius e Clock-RSM, e serão discutidos na Seção 2.9.

2.9 Trabalhos relacionados

Nesta Seção apresentaremos alguns trabalhos com objetivos semelhantes ao nosso, no que tange uma avaliação teórica e experimental de protocolos de acordo, que é o principal objetivo desse trabalho.

Além disso, também exemplificaremos trabalhos que tentam esclarecer o funcionamento de protocolos, no intuito de facilitar suas implementações em sistemas reais, que é um dos objetivos secundários desse trabalho.

2.9.1 Paxos Made Live

Aplicar na prática uma teoria nem sempre é fácil, ainda mais quando se trata de algoritmos para tolerância a falhas. Especificações de protocolos são normalmente fornecidas de forma concisa (e.g. pseudo-código ou linguagens formais de especificação, como TLA+), muitas das vezes englobando apenas aspectos teóricos e sendo aplicáveis a um limitado conjunto de cenários e falhas.

Na prática, implementar tais protocolos em um sistema real, que está exposto a uma variedade de modelos de falhas, inclusive falhas de implementação (i.e. *bugs*) e mesmo de operação, demanda a criação de mecanismos auxiliares omitidos na especificação do protocolo original, por exemplo, a definição de um algoritmo de eleição de Líder. Tais mecanismos podem impactar diretamente nas garantias teóricas oferecidas pelo protocolo, assim como no desempenho de um sistema replicado que o utilize.

Em (CHANDRA; GRIESEMER; REDSTONE, 2007) é descrito um desses esforços, de se construir um sistema replicado tolerante a falhas utilizando *Paxos*. No trabalho, foi necessário o uso de mecanismos auxiliares para se conseguir desenvolver um sistema gerenciador de bancos de dados replicado, com garantia de consistência forte e que apresenta desempenho competitivo com soluções existentes.

Nosso trabalho também tem esse cunho exploratório e avaliativo com relação ao desempenho, porém com o foco no uso do protocolo *Collision-fast Paxos* para replicação de máquinas de estado. Durante o desenvolvimento do *Collision-fast Atomic Broadcast* enfrentamos problemas similares aos descritos em (CHANDRA; GRIESEMER; REDSTONE,

2007), como por exemplo, a necessidade de se criar um módulo para tratar mudanças no conjunto de réplicas (*Group Membership*). Essa e outras funcionalidades, necessárias para a criação de um sistema real que utilize o protocolo proposto em (SCHMIDT; CAMARGOS; PEDONE, 2007), mas omitidas na especificação do protocolo, estão descritas na Seção 3 desse trabalho.

2.9.2 Paxos for System Builders

Outro trabalho que tenta descrever de forma clara e objetiva uma especificação de um protocolo de acordo é o (KIRSCH; AMIR, 2008), no qual Kirsch e Amir apresentam uma especificação do *Paxos* que contempla detalhes sobre quais mecanismos utilizar para se detectar falhas ou eleger um Líder, de modo a facilitar o uso do *Paxos* em sistemas reais.

Assim como em (KIRSCH; AMIR, 2008) também almejamos objetivo similar, com o intuito de demonstrar à comunidade científica os benefícios de se utilizar o *Collision-fast Paxos* na resolução do *M-Consensus* e como este protocolo pode ser utilizado na implementação de um sistema replicado que seja competitivo em termos de desempenho com os sistemas atuais.

2.9.3 Multi-Ring Paxos

Multi-Ring Paxos (MARANDI; PRIMI; PEDONE, 2012) é um protocolo de *multicast* atômico que utiliza múltiplas instâncias de *Ring Paxos* (MARANDI et al., 2010) na tentativa de aumentar a vazão sem sacrificar a latência em um sistema replicado.

No *Ring Paxos* os *acceptors* são dispostos em um anel, sendo o Líder um elemento desse anel. Ao receber uma nova proposta, cada *acceptor* a acrescenta na sua lista de aceites e a envia para seu vizinho. Em um dado momento, essa mensagem chega ao *learner* que esta associado ao grupo, que consegue contabilizar qual decisão foi tomada.

Um atraso na entrega das mensagens em um dos participantes do anel afeta os demais participantes, e pode acabar por prejudicar o desempenho do sistema, por essa razão, tal protocolo é muito eficiente em redes locais onde a confiança na correção dos nós é maior, e tais problemas são raros.

No *Multi-Ring Paxos*, cada instância do *Ring Paxos* está associada a um grupo de comunicação. O *multicast* atômico garante que as mensagens serão entregues de forma ordenada para todos aqueles que receberem as mesmas mensagens do grupo. Quando uma instância do *Ring Paxos* está associada a mais de um grupo, o protocolo faz uso de um mecanismo de junção determinístico para garantir que todas as mensagens serão entregues em uma mesma ordem.

Este trabalho utilizou uma implementação do protocolo *Multi-Ring Paxos* disponível na biblioteca *open source* denominada Multicast Adaptor Library (Libmcad) [1], realizando

[1] <https://bitbucket.org/kdubezerra/libmcad>

multicast atômico para apenas um grupo de servidores, com o intuito de comparar a nossa implementação com uma mais robusta e já amplamente testada em condições reais.

2.9.4 Mencius

Mencius (MAO; JUNQUEIRA; MARZULLO, 2008) é um protocolo para replicação de máquinas de estados baseado em uma modificação do protocolo Paxos, que resolve uma versão simplificada do problema de Consenso. Em cada instância do protocolo, um servidor é designado como coordenador, e apenas o coordenador pode propor qualquer valor, incluindo um valor especial *no-op* que deixa o estado inalterado; os demais servidores podem propor apenas *no-op*. Múltiplas instâncias são realizadas em paralelo para se chegar a um acordo em um conjunto de mensagens a serem entregues, de modo similar ao que é feito pelo protocolo *Collision-Fast Paxos*.

Na ausência de falhas, ambos protocolos possuem um padrão de comunicação similar. Porém, no Mencius, se um dos coordenadores falhar, os demais deverão continuar propondo *no-op* nas instâncias desse coordenador. Já no protocolo *Collision-Fast Paxos* se um collision-fast proposer falhar, o coordenador pode reconfigurar as instâncias de *M-Consensus* com um conjunto diferente de *collision-fast proposers* para a próxima rodada, permitindo que a execução continue *collision-fast*. O coordenador é escolhido usando um oráculo de eleição de Líderes do tipo Ω (CHANDRA; HADZILACOS; TOUEG, 1996).

Pelo fato do protocolo *Collision-Fast Paxos* conseguir se manter *collision-fast* após reconfigurações decorrentes de falhas, diferentemente do Mencius que precisa ser reinicializado ou continuar trocando mensagens de controle para tratar processos faltosos, entende-se que este último não é de fato *collision-fast* (SCHMIDT; CAMARGOS; PEDONE, 2007).

2.9.5 Clock-RSM

Clock-RSM (DU et al., 2014) também é um protocolo para replicação de máquinas de estados que provê baixa latência, e faz uso de relógios físicos fracamente sincronizados (por um protocolo de sincronização de tempo, como por exemplo o NTP) em cada réplica para ordenar totalmente os comandos. O padrão de comunicação do Clock-RSM é semelhante ao do *Collision-Fast Paxos*, se todos os *proposers* propuserem com frequência constante. Além disso, ambos possuem comportamentos semelhantes, como por exemplo, o Clock-RSM requer que uma réplica receba de uma maioria de servidores mensagens *PrepareOK* para aprender algum comando, assim como o *learner* no protocolo *Collision-Fast Paxos* requer que um comando seja aceito por um *quorum* de *acceptors*.

Diferentemente do protocolo *Collision-Fast Paxos*, em que a reconfiguração das instâncias futuras é realizada pelo coordenador a cada nova rodada sem a necessidade de Consenso, no Clock-RSM a reconfiguração é feita resolvendo-se uma instância de Con-

senso explicitamente, o que implica em eleição de um Líder e não garantia de terminação. Contudo, a maior diferença entre os protocolos é que o foco do Clock-RSM é o resultado final, a replicação de máquinas de estados, enquanto que no *Collision-Fast Paxos* a preocupação é com o meio, o algoritmo de Difusão Atômica.

Algoritmo 2.1 Collision-fast Paxos

Pr: proposers set; *A*: acceptors set; *L*: learners set;

$CF(i)$: round i's collision-fast proposers set; $C(i)$: round i's coordinator.

$prnd[p]$, $crnd[c]$, $rnd[a]$: current round of proposer p, coordinator c, and acceptor a, respectively, initially 0.

$pval[p]$: value p has fast-proposed at $prnd[p]$ or *none* if p has not fast-proposed at $prnd[p]$, initially *none*.

$cval[c]$: initial v-mapping for $crnd[c]$, if c has queried an acceptor quorum or *none* otherwise; initially $\bot$ for coordinator of round 0 and *none* for others.

$vrnd[a]$: round at which a has accepted its latest value.

$vval[a]$: v-mapping a has accepted at $vrnd[a]$ or *none* if no value accepted at $vrnd[a]$; initially *none*.

learned$[l]$: v-mapping currently learned by learner l; initially $\bot$.

1: *Propose*(p, V) $\triangleq$
2: **p**re-condition:
3: $p \in Pr$
4: **a**ction:
5: send $\langle$"`propose`", $V\rangle$ to $cf \in CF(prnd[p])$
6: *Phase1a*(c, r) $\triangleq$
7: **p**re-conditions:
8: $c = C(r)$
9: $crnd[c] < r$
10: **a**ctions:
11: $crnd[c] \leftarrow r$
12: $cval[c] \leftarrow none$
13: send $\langle$"`1a`", $r\rangle$ to A
14: *Phase1b*(a, r) $\triangleq$
15: **p**re-conditions:
16: $a \in A$
17: $rnd[a] < r$
18: received $\langle$"`1a`", $r\rangle$ from $C(r)$
19: **a**ctions:
20: $rnd[a] \leftarrow r$
21: send $\langle$"`1b`", $a, r, vrnd[a], vval[a]\rangle$ to $C(r)$
22: *Phase2Start*(c, r) $\triangleq$
23: **p**re-conditions:
24: $c = C(r)$
25: $crnd[c] = r$
26: $cval[c] = none$
27: $\exists Q : Q$ is a quorum: $\forall a \in Q$ received $\langle$"`1b`", $a, r, vrnd, vval\rangle$
28: **a**ctions:
29: LET $msgs = [m = \langle$"`1b`", $a, r, vrnd, vval\rangle$: received m from $a \in Q\ ]$)
30: LET $k = Max([vrnd : \langle$"`1b`", $a, r, vrnd, vval\rangle \in msgs\ ])$
31: LET $S = [vval : \langle$"`1b`", $a, r, k, vval\rangle \in m\ , vval \neq none]$
32: IF $S = \emptyset$ THEN
33: $cval[c] \leftarrow \bot$
34: send $\langle$"`2S`", $r, cval[c]\rangle$ to P
35: ELSE
36: $cval[c] \leftarrow \sqcup S \bullet [\langle p, Nil\rangle : p \in Pr]$
37: send $\langle$"`2S`", $r, cval[c]\rangle$ to $P \cup A$
38: *Phase2Prepare*(p, r) $\triangleq$
39: **p**re-conditions:
40: $p \in Pr$
41: $prnd[p] < r$
42: received $\langle$"`2S`", $c, r, v\rangle$
43: **a**ctions:
44: $prnd[p] \leftarrow r$
45: IF $v = \bot$ THEN $pval[p] \leftarrow none$ ELSE $pval[p] \leftarrow v(p)$
46: *Phase2a*(p, r, V) $\triangleq$
47: **p**re-conditions:
48: $p \in CF(r)$
49: $prnd[p] = r$
50: $pval[p] = none$
51: **either** $V \neq Nil$ **and** received $\langle$"`propose`", $p, V\rangle$
52: **or** $V = Nil$ **and** received $\langle$"`2a`", $r, \langle q, W\rangle\rangle$, $q \in CF(r)$, $W \neq Nil$
53: **a**ctions:
54: $pval[p] \leftarrow V$
55: **if** $V \neq Nil$ **then** send $\langle$"`2a`", $p, r, \langle p, V\rangle\rangle$ to $A \cup CF(r)$
56: **else** send $\langle$"`2a`", $p, r, \langle p, V\rangle\rangle$ to L
57: *Phase2b*(a, r) $\triangleq$
58: LET *Cond1* = (received $\langle$"`2S`", $r, v\rangle$, $v \neq \bot$ **and** $vrnd[a] < r$) **or** $vval[a] = none$
59: LET *Cond2* = received $\langle$"`2a`", $r, \langle p, V\rangle\rangle$, $V \neq Nil$
60: **p**re-conditions:
61: $a \in A$
62: $rnd[a] \leq r$
63: **either** *Cond1* **or** *Cond2*
64: **a**ctions:
65: $rnd[a] \leftarrow vrnd[a] \leftarrow r$
66: IF *Cond1* THEN $vval[a] \leftarrow v$
67: ELSE IF *Cond2* **and** ($vrnd[a] < r$ **or** $vval[a] = none$)
68: THEN $vval[a] \leftarrow \bot \bullet \langle p, V\rangle \bullet [\langle p, Nil\rangle : p \in Pr \setminus CF(r)]$
69: ELSE $vval[a] \leftarrow vval[a] \bullet \langle p, V\rangle$
70: send $\langle$"`2b`", $a, r, vval[a]\rangle$ to L
71: *Learn*(l) $\triangleq$
72: **p**re-conditions:
73: $l \in learners$
74: $\exists Q, Q$ is a quorum: $\forall a \in Q$ received $\langle$"`2b`", $a, r, \_\rangle$
75: **a**ctions:
76: LET $P \subset CF(r) : \forall p \in P$ received $\langle$"`2a`", $p, r, \langle p, Nil\rangle\rangle$
77: $Q2bVals = [v :$ received $\langle$"`2b`", $a, r, v\rangle$ from $a \in Q\ ]$
78: $w = \sqcap Q2bVals \bullet [\langle p, Nil\rangle : p \in P]$
79: $learned[l] = learned[l] \sqcup w$

Algoritmo 2.2 Collision-fast Atomic Broadcast

I: the set of all Collision-fast Paxos instances used
$CFP(i)!A$: the action or variable A of Collision-fast Paxos instance i

1: $Propose(p, V) \triangleq$
2: $\forall i \in I, CFP(i)!Propose(p, V)$
3: $NewPhase1a(i, c, r) \triangleq$
4: **pre-condition:**
5: $c = C(r)$
6: $crnd[c] < r$
7: c believes itself to be the leader
8: c heard of a round $r > j > crnd[c]$ for some instance or $CF(crnd[c]) \notin active[c]$
9: **actions:**
10: $CFP(i)!Phase1a(c, r)$
11: $Phase1a(c, r) \triangleq$
12: $\forall i \in I, CFP(i)!NewPhase1a(i, c, r)$
13: $Phase1b(a, r) \triangleq$
14: $\forall i \in I, CFP(i)!Phase1b(a, r)$
15: $Phase2Start(c, r) \triangleq$
16: $\forall i \in I, CFP(i)!Phase2Start(c, r)$
17: $Phase2Prepare(p, r) \triangleq$
18: $\forall i \in I, CFP(i)!Phase2Prepare(p, r)$
19: $Phase2a(p, r, V) \triangleq$
20: **pre-condition:**
21: p has not yet proposed V
22: **action:**
23: LET $i = Min([j : CFP(j)!pval[p] = none])$
24: $CFP(i)!Phase2a(p, r, V)$
25: $Phase2b(i, a, r) \triangleq$
26: $CFP(i)!Phase2b(a, r)$
27: $Learn(i, l) \triangleq$
28: $CFP(i)!Learn(l)$

Capítulo 3

Collision-fast Atomic Broadcast

O algoritmo *Collision-fast Atomic Broadcast* foi implementado como parte de um *framework* para replicação de máquinas de estados, cujo objetivo, além de permitir a avaliação deste protocolo, é servir de ambiente modular para avaliação de protocolos de acordo. No nosso *framework* o *Collision-fast Paxos* foi utilizado para garantir a ordem total de execução das requisições recebidas em cada réplica de um serviço genérico, descrito mais adiante. A Figura 8 ilustra o fluxo principal de comunicação do nosso *framework*.

Na Figura 8, as aplicações são as réplicas. Os clientes acessam essas réplicas emitindo comandos, que são repassados para o nível de *Atomic Broadcast*, que entrega a todas as réplicas da aplicação os comandos na mesma ordem garantindo consistência.

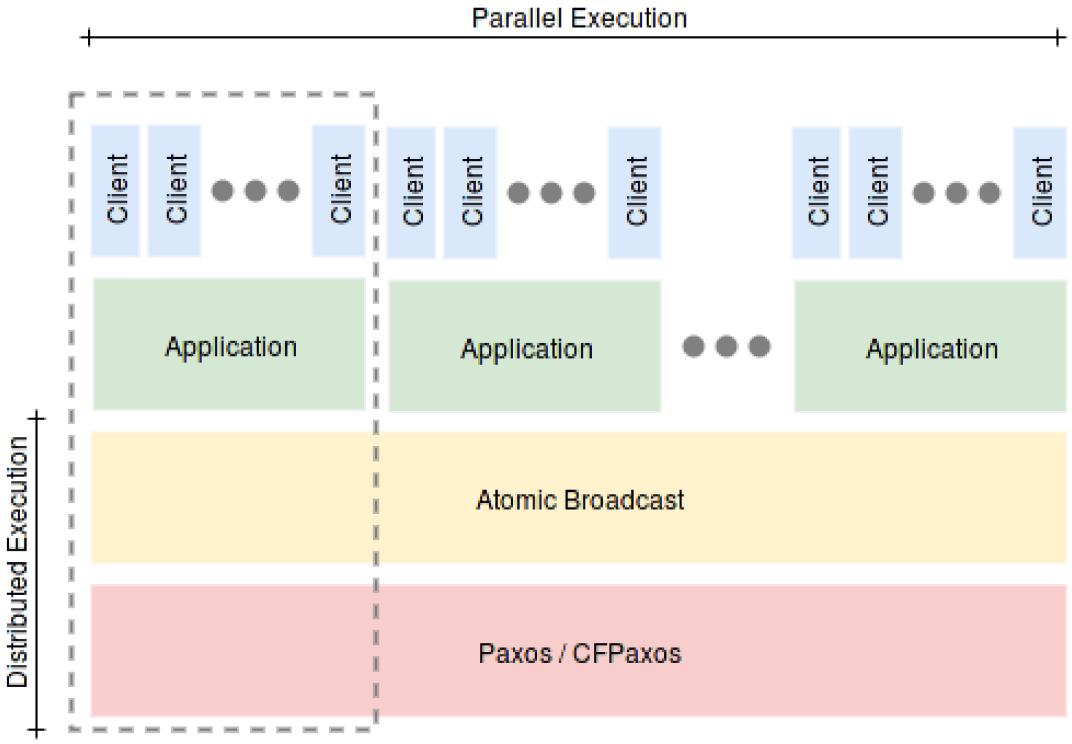


Figura 8 – Níveis de comunicação no *framework* desenvolvido

Neste capítulo descreveremos as decisões tomadas durante o desenvolvimento desse

framework para replicação de máquinas de estado.

3.1 Arquitetura

Embora o protocolo *Collision-fast Paxos* seja especificado em TLA+ de forma concisa, assim como é feito com vários outros algoritmos baseados no *Paxos*, a tradução de especificação para uma implementação confiável e de alto desempenho é notoriamente difícil, como observado em outros trabalhos (CHANDRA; GRIESEMER; REDSTONE, 2007) e (KIRSCH; AMIR, 2008).

Além disso, durante a implementação do algoritmo *Collision-fast Paxos* foi necessário resolver uma série de problemas práticos decorrentes das abstrações e suposições contidas na descrição do algoritmo, que por sua vez deve se manter simples em prol da corretude. Muitas das escolhas de *design* apenas se tornaram evidentes após nos depararmos com algum problema que não estava explicito na especificação do protocolo, às vezes sendo necessário refazer parte do sistema.

A forma como esses problemas são resolvidos na implementação impacta diretamente no desempenho do algoritmo e da aplicação que o utiliza, e por esta razão devem ser levadas em consideração durante a etapa de projeto.

3.1.1 Tecnologias utilizadas

O protocolo implementado neste trabalho foi escrito na linguagem de programação Scala, que mistura conceitos de linguagens orientada a objetos com linguagens funcionais, fazendo uso de princípios como imutabilidade, funções de alta ordem, *closures*, etc, comuns em linguagens funcionais mas sem perder os benefícios de uma linguagem puramente orientada a objetos. Além de oferecer de uma forma concisa, elegante e *type-safe*, as mesmas funcionalidades que a linguagem Java, mantém interoperabilidade com esta e executa sobre Java Virtual Machine (JVM).

Ademais, utilizamos um conjunto de ferramentas conhecido como *Akka*, que dispõe de funcionalidades que auxiliam na construção de sistemas distribuídos, adotando o modelo de atores para concorrência ao invés do tradicional uso de *threads* e *locks*.

3.1.1.1 Porque não usar *threads*?

A forma tradicional de se oferecer processamento paralelo ou concorrente é utilizando-se *threads*. Neste modelo a execução do programa é dividida em linhas de execução concorrentes, com acesso uma memória compartilhada.

Isto pode levar a uma série de problemas difíceis de depurar, como por exemplo o problema de *lost-update*. Suponha que dois processos estão tentando incrementar o valor de uma variável compartilhada *acc*: ambos recuperam o valor da variável, incrementam

e o armazenam novamente na variável compartilhada. Como estas operações não são atômicas, é possível que suas execuções sejam intercaladas, acarretando em atualizações incorretas no valor de *acc*.

Para solucionar tal problema, usa-se um mecanismo de sincronização conhecido como *lock*, que provê uma política de controle de concorrência de exclusão mútua, ou seja, apenas um processo por vez pode acessar o recurso compartilhado. Porém o uso de *locks* pode tornar a implementação mais complexa e acarreta em muitos outros problemas, como por exemplo *deadlocks*, quando dois processos tentam adquirir os mesmos dois *locks* em diferente ordem, ambos esperarão indefinidamente a liberação dos mesmos.

De forma geral, acesso concorrente à dados compartilhados utilizando mecanismos de sincronização tende a ser propenso a erros e de difícil depuração (BUTCHER, 2014). Existem outros modelos de concorrência que são mais fáceis de entender e depurar que o tradicional, como o modelo de atores, que oferecem melhor suporte para desenvolvimento de sistemas distribuídos e tolerante a falhas.

3.1.1.2 Modelo de atores

O modelo de atores, inicialmente proposto por Carl Hewitt em 1973 (HEWITT; BISHOP; STEIGER, 1973), utiliza uma abordagem diferente para tratar concorrência, que ajuda a evitar os problemas causados pelo uso de *threads* e *locks*.

Neste modelo, o processamento é realizado por "pequenos" processos independentes (i.e. de baixo custo computacional - *lightweight threads*) denominados *atores*, que possuem uma caixa de correio (*mailbox*) para armazenamento de mensagens e um comportamento pré-definido que pode se alterar durante execução.

Cada ator executa de forma concorrente com outros, e toda a comunicação é feita por troca de mensagens, não havendo compartilhamento de estado entre os atores. Além disso, a comunicação é assíncrona, não tendo garantia na ordem de entrega das mensagens, mas as entregando em algum momento.

O conceito de atores, fornece um modelo ideal para desenvolver sistemas altamente concorrente e escaláveis. Implementar um sistema concorrente usando os métodos de baixo nível tradicionais, como *locks* e *threads* pode ser um trabalho árduo, pois o software resultante muitas das vezes é mais difícil de entender e manter.

Desta forma, o modelo de atores fornece uma alternativa de alto nível de abstração, que é muito mais fácil de se trabalhar do que o tradicional. Os atores também fornecem transparência de localização, ou seja, o usuário de um ator não precisa saber se esse ator reside no mesmo processo ou em uma máquina diferente para se comunicar com ele, tornando mais simples a tarefa de se implementar softwares escaláveis de acordo com a necessidade da aplicação.

Originalmente concebida pela Ericsson, Erlang é uma linguagem de programação funcional que foi especialmente desenvolvida para sanar os problemas existentes em aplicações

de redes de grande escala, altamente concorrentes, onde há necessidade de baixo tempo de respostas e alta disponibilidade.

Erlang utiliza o modelo de atores para fornecer suporte a construção de sistemas distribuídos e tolerante a falhas, sendo amplamente utilizada em aplicações de sistemas de tempo real, como na indústria de telecomunicações.

3.1.1.3 Akka

Este trabalho utiliza como base para implementação do protocolo *Collision-fast Atomic Broadcast* o *middleware* Akka[1], um arcabouço *open-source* para a construção de sistemas concorrentes, distribuídos e tolerantes a falhas na JVM. Akka foi escrito na linguagem Scala, mas oferece APIs em Scala e Java, e suporte para vários modelos de programação concorrente, porém enfatiza o uso do modelo baseado em atores para concorrência, inspirado no modelo adotado pela linguagem Erlang.

Akka propicia aos desenvolvedores uma forma unificada de construir um software escalável e tolerante a falhas que pode escalar tanto verticalmente (*scale up*) em sistemas com múltiplos *cores* como horizontalmente (*scale out*) em ambientes de computação distribuída. Desta forma é possível construir sistemas com confiabilidade e alto desempenho, sem sacrificar o desenvolvedor e sua produtividade, como um sistema de replicação de máquinas de estado.

O modelo de tolerância a falhas adotado pelo Akka é inspirado no adotado pela linguagem de programação Erlang, comumente conhecido como "*let it crash*". Neste modelo, os atores são organizados em uma hierarquia supervisionada, de modo que cada componente é monitorado por um outro, podendo ser reiniciado em caso de falhas. O modelo tem sido usado com sucesso na indústria de telecomunicações há anos, possibilitando a criação de sistemas altamente disponíveis que podem se curar de forma adaptativa durante sua execução.

A comunicação entre os atores é baseada em troca de mensagens de forma assíncrona e normalmente os dados compartilhados não são mutáveis,em geral, não havendo necessidade de uso de primitivas de sincronização . Além disso, Akka possui uma estrutura modular que possibilita maior flexibilidade na construção de um sistema distribuído, tendo os atores como módulo principal.

Os demais módulos estão disponíveis como funcionalidades extras, e alguns deles são utilizados nesse trabalho, tais como i)interface com protocolos TCP/UDP, que fornece uma abstração de alto nível para comunicação entre atores e transparência de localização e, ii)suporte a clusterização, que oferece ferramentas para gerenciamento de grupos de atores distribuídos em um *cluster* e mecanismos de detecção de falhas. Além disso, há outros módulos que permitem integração com diversos sistemas terceiros (como por exemplo

[1] <http://akka.io/>

ZeroMQ) e que podem servir de pontos de extensão de nosso protótipo em trabalhos futuros.

Finalmente, Akka possui também um sistema de *Futures* e *Promises*, que permitem o uso do paradigma de programação assíncrona, e que também é explorado nesse trabalho (ver Seção 3.1.4.1) como modelo complementar ao modelo de atores, para se conseguir maior paralelismo em cada agente.

3.1.2 Projeto

O sistema implementado é composto por um conjunto de entidades (Akka cluster), aqui denominados nós. Cada nó do sistema possui um endereço único para comunicação e desempenha o papel de uma réplica para a aplicação final, cuja função é manter uma cópia dos dados da aplicação de modo a aumentar a confiabilidade, tolerância a falhas e acessibilidade do sistema.

Uma réplica, por sua vez, é composta por vários módulos, cada um responsável por uma função no sistema, e uma interface para comunicação com uma aplicação, como mostrado na Figura 9.

Inicialmente um cliente estabelece uma conexão com alguma das réplicas do sistema. Esta, registra o módulo cliente da aplicação e o associa a um agente que desempenha o papel de *proposer* no protocolo. O mesmo é feito para um módulo servidor da aplicação que queira saber o que foi aprendido em alguma instância do protocolo, porém, ao invés de associar-se a um *proposer*, é associado a um *learner*. Isto é feito de forma a distribuir os clientes e servidores entre os nós do *cluster* seguindo uma política *Round Robin*.

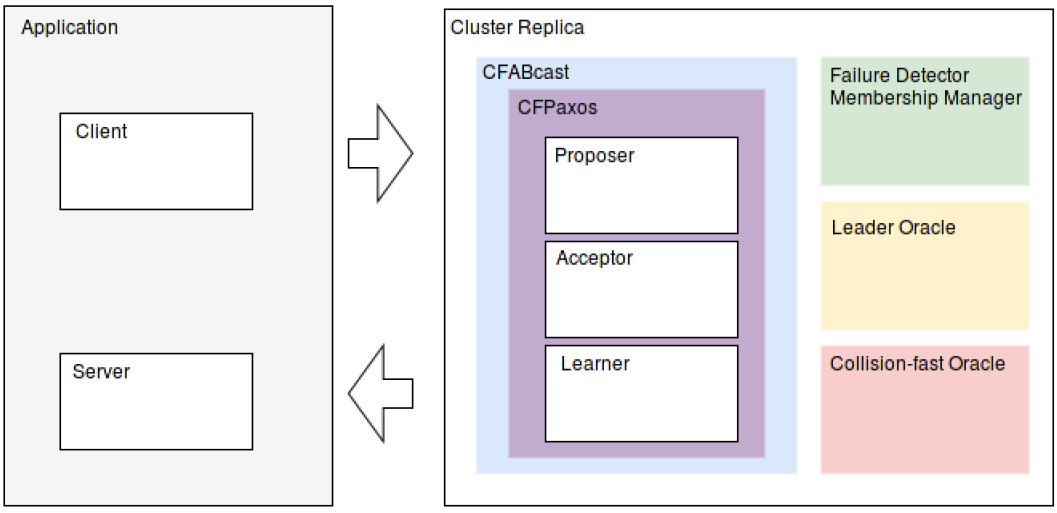


Figura 9 – Arquitetura modular dos principais componentes de uma réplica no nosso *framework* e sua interface com uma aplicação

Cada agente do protocolo *Collision-fast Paxos* (*proposers*, *acceptors*, *learners*), assim como cada módulo pertencente a uma réplica, inclusive a própria réplica, são implementados como personificações de atores do Akka, processos leves que podem ser criados

aos milhares por réplica e comunicam-se assincronamente por troca de mensagens para resolver o problema de *M-Consensus*.

Esses atores são supervisionados pela própria réplica, que em conjunto com o módulo de gerenciamento de membros do *cluster* e detector de falhas, permite a reconfiguração e substituição de réplicas/agentes falhos durante a execução do protocolo.

A seguir, descreveremos os principais módulos criados no nosso *framework*, desenvolvidos como suporte ao protocolo implementado, possibilitando seu uso na criação de uma aplicação de replicação de máquinas de estados tolerante a falhas.

3.1.2.1 *Membership Manager*

Este módulo é responsável por tratar alterações no conjunto de réplicas, que é um problema conhecido na literatura como *Group membership* (CRISTIAN, 1991). Alguns exemplos que envolvem tal problema em aplicações reais são: descobrimento de novas réplicas ou remoção de réplicas suspeitas de falha.

Existem várias formas de resolvê-lo, como por exemplo, utilizando o próprio protocolo que resolve *consensus*, como o *Paxos* (LAMPORT, 1998). Porém na prática o conjunto de réplicas não sofre alterações frequentes, e um protocolo com melhor desempenho mas consistência eventual poderia ser utilizado. Nesse trabalho, o problema de gerenciamento de grupos é resolvido utilizando-se um protocolo de "fofoca" (do inglês *gossip*) oferecido pelo Akka. Esses protocolos tem se mostrado eficazes na detecção de falhas e gerenciamento de membros em sistemas assíncronos, distribuídos e de grande escala (SUBRAMANIYAN et al., 2006). A abordagem é utilizada por uma gama de aplicações, como o *Amazon Dynamo*, um sistema de armazenamento de chave-valor de alta disponibilidade que utiliza de *gossip* no protocolo de detecção de falhas e controle dos nós do *cluster*. (DECANDIA et al., 2007).

Tais protocolos, diferente do *Paxos*, oferecem consistência eventual e são conceitualmente simples. Basicamente, cada nó periodicamente envia alguns dados para um conjunto de outros nós vizinhos, escolhidos aleatoriamente. Os dados se propagam pelo sistema nó a nó, semelhante a uma epidemia de vírus, em algum momento atingindo todos os nós do sistema. Nesse ponto, o protocolo converge, sendo possível que os nós construam um mapa global de informações do sistema com base nessas trocas de dados entre vizinhos. Assim, a informação trocada pode ser usada tanto para fornecer dados de saúde de cada nó ao detector de falhas, como também para implementar métodos de descobrimento de nós no sistema, dentre outras coisas, como mostrado em (EUGSTER et al., 2004).

O diagrama de estados representado na Figura 10 ilustra todos os estados possíveis de cada nó de um *cluster* durante a execução do sistema. Como mostrado na figura, inicialmente os nós do *cluster* (réplicas no protocolo) estão no estado *Joining* e nesse

estado é executada uma fase de descoberta dos membros do *cluster* utilizando o protocolo de *gossip*.

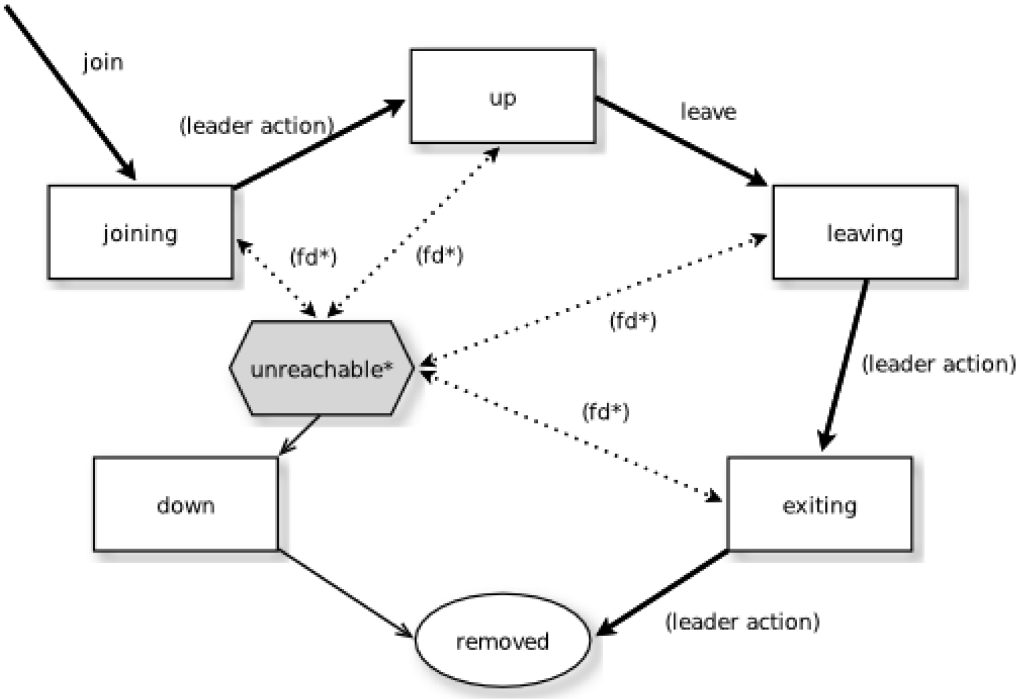


Figura 10 – Diagrama de estados de um nó do cluster

Fonte: http://doc.akka.io/docs/akka/2.4.3/common/cluster.html#Membership_Lifecycle

Uma vez que todos os nós viram que um novo nó esta se juntando ao *cluster* (no estado de *Joining*), o estado do nó é alterado para *Up*. Assim, em algum momento o *cluster* alcança o tamanho configurado inicialmente, e os nós iniciam a fase de eleição de líder usada no protocolo *Collision-fast Paxos*, em que o módulo de Eleição de líder é notificado.

Um nó pode ser removido do cluster de duas formas: terminando sua execução normalmente e mudando para o estado *Leaving* e depois *Exiting* e finalmente sendo removido alterando para o estado *Removed*; ou, tendo seu estado alterado para *Unreachable* pelo detector de falhas (transições mostradas por (fd*) na Figura 10) e posteriormente para *down*, caso não seja detectada novamente sua alcançabilidade.

Quando um ator é detectado como *Unreachable* pelo detector de falhas, este notifica o módulo de controle de membros, que após período de sincronia, remove o ator do sistema e atualiza a configuração dos membros do sistema, disseminando-a pela rede e notificando o módulo de eleição de líder que uma nova eleição pode ser necessária. De posse dessa nova configuração de membros os agentes atualizam seu estado e se um novo líder for eleito, executam a etapa de reconfiguração do protocolo (*Phase1*).

3.1.2.2 *Failure Detector*

Chandra e Toueg (CHANDRA; TOUEG, 1995) introduziram o conceito de Detectores de Falhas Não-Confiáveis. Um detector de falhas é um oráculo distribuído acoplado a cada nó do *cluster* que opera de forma a determinar o estado funcional desse nó e dos seus demais componentes, como mostrado na Figura 9.

No nosso *framework* o módulo de gerenciamento de membros também é responsável por tentar detectar falhas durante a execução do protocolo, e foi utilizada a implementação do Akka do detector de falhas conhecido como *Phi Accrual Failure Detector* (DÉFAGO et al., 2004).

Esse detector de falhas mantém um histórico de estatísticas de falhas calculadas a partir de mensagens periódicas (*heartbeats*) trocadas entre os nós. Dessa forma, ele tenta supor, com base nos múltiplos dados acumulados ao longo do tempo, o estado do nó entre *Up* ou *Down*, como mostrado na Figura 10. O que resulta em uma probabilidade do nó estar falho, sendo marcado como *unreachable* pelo detector de falhas.

O nível de suspeita de falha de um nó é dado por um valor denominado ϕ, calculado pela fórmula apresentada na equação 2. A ideia básica desse detector de falhas é expressar o valor de *Phi* em uma escala dinamicamente ajustável no intuito de refletir as condições atuais da rede. Esse limite configurável é utilizado para decidir se ϕ é considerado uma falha ou não.

O uso de um limite baixo propicia a geração de falsos positivos, mas garante uma detecção rápida em casos de falhas reais. Por outro lado, um limite alto gera menos erros, mas precisa de mais tempo para detectar falhas reais. Tal limite pode ser alterado através da variável: `akka.cluster.failure-detector.threshold` existente no arquivo de configuração do protótipo.

$$\phi = -log_{10}(1 - F(timeSinceLastHeartbeat)) \tag{2}$$

Na Equação 2, F é a função de distribuição cumulativa de uma distribuição normal com média e desvio padrão estimados a partir de um histórico de intervalos de tempos de chegada de mensagens de *heartbeats*.

As mensagens de *heartbeats* no Akka são disseminadas pelo protocolo de fofoca (*Gossip*), que tem como objetivo principal espalhar uma informação que em algum momento atinge todos os membros da rede.

Esse protocolo fornece um grau de tolerância a falhas superior a de outros protocolos, em detrimento do grau de consistência do sistema. Tal relação já é bem conhecida na literatura, graças ao Teorema Consistency, Availability and Partition tolerance (CAP), que estabelece que é impossível para um sistema distribuído fornecer simultaneamente todas as três propriedades (GILBERT; LYNCH, 2002).

- ❏ Consistência: Todos os nós veem os mesmos dados ao mesmo tempo, como se existisse apenas uma cópia sempre mais atual dos dados visível a todos os nós no mesmo instante.

- ❏ Disponibilidade: Os dados estão sempre disponíveis para leitura e atualização (Alta disponibilidade).

- ❏ Tolerância à particionamento: o funcionamento do sistema independe de perda de conexão entre nós do sistema devido particionamento arbitrário por falhas na rede.

O teorema auxilia no projeto de protocolos, possibilitando um melhor ponderamento a respeito de qual protocolo é ideal para cada problema, como ilustrado na Figura 11, por exemplo, protocolos de *consensus* como o *Paxos*, garantem consistência forte pois requerem que um *quorum* de *acceptors* concordem em um dado valor antes de entrega-lo à aplicação, ou seja, uma maioria dos nós sempre terão uma cópia mais atual de um dado valor. Porém se um particionamento da rede à divide em n *acceptors*, então a parte da rede que não se comunica com $n/2$, considerando um *quorum* pela maioria, não conseguirá formar um *quorum* e portanto não chegará à uma decisão, consequentemente, não entregando o valor para a aplicação. Assim tais nós não estarão disponíveis para a aplicação final que utiliza do *Paxos*.

Por outro lado, o particionamento em uma rede que utilize um protocolo de fofoca, não tem seu progresso prejudicado como o exemplo anterior, pois cada partição continua disseminando mensagens entre si. Porém, uma requisição feita a uma partição pode obter uma resposta totalmente diferente se atingir a outra partição, prejudicando assim a consistência do protocolo. Isso possibilita que protocolos de fofoca sejam comumente utilizados para construção de detectores de falhas, como é o caso da implementação utilizada no módulo *Akka Cluster* e de trabalhos como (RENESSE; MINSKY; HAYDEN, 1998) e (WARD; BARKER, 2014)

No *cluster* cada nó é monitorado por uma quantidade configurável de outros nós, utilizando as mensagens de *heartbeats* e quando algum nó detecta um outro como *inacessível*, essa informação é espalhada para os demais nós do *cluster* através de um protocolo de *Gossip*.

Em outras palavras, é necessário que apenas um nó suspeite que algum está inacessível para que os demais também suspeitem. As informações de tempo de chegada das mensagens de *heartbeats* são interpretadas pelo detector de falhas. De forma similar, um nó pode ser detectado acessível novamente pelo mesmo protocolo de fofoca, porém requer que um grupo de nós que monitoram o suspeito o detectem e disseminem essa informação.

Os módulos de eleição de líder, seleção de *Collision-fast proposers* e controle de membros fazem uso da heurística do detector de falhas para tomar decisões no nível do protocolo de *Atomic Broadcast* implementado neste trabalho, permitindo assim a reconfiguração do sistema e sua re-estabilização após alguma falha.

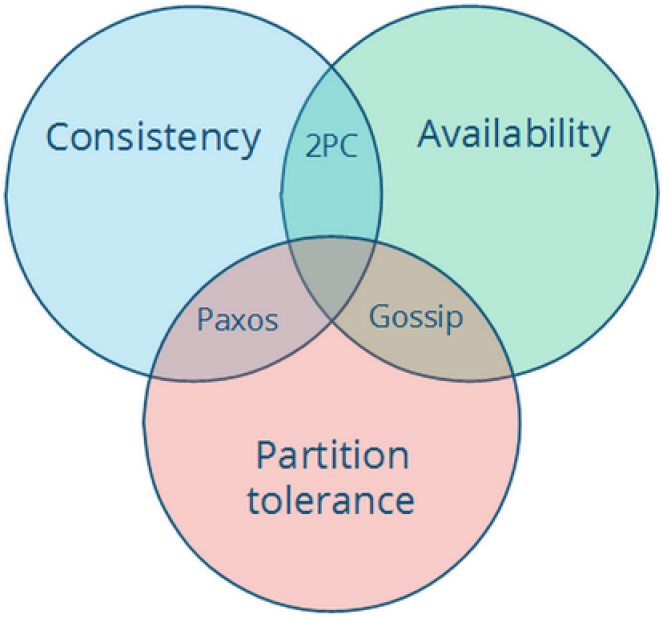


Figura 11 – CAP

Fonte: https://en.wikipedia.org/wiki/CAP_theorem

3.1.2.3 Líderes e Rodadas

Um dos resultados mais importantes da área de teoria em Sistemas Distribuídos foi publicado em 1985 por Fischer, Lynch e Patterson e é conhecido como o resultado FLP. Em (FISCHER; LYNCH; PATERSON, 1985) os autores provam que é impossível resolver *Consensus* em um sistema totalmente assíncrono onde um ou mais agentes (que desempenhe a função de *acceptor*) possam falhar. Tal trabalho impulsionou uma série de pesquisas na comunidade teórica de sistemas distribuídos, como, por exemplo, o aparecimento de detectores de falhas (CHANDRA; HADZILACOS; TOUEG, 1996) na tentativa de contornar tal impossibilidade, mensurando o nível de assincronismo mínimo para se resolver *Consensus*.

Em (CHANDRA; HADZILACOS; TOUEG, 1996), Chandra e Toueg demonstraram que o detector de falhas mais fraco que pode ser usado para se contornar a FLP é o oráculo de eleição de líderes Ω. Logo, assim como no protocolo *Paxos* proposto por Lamport, o *Collision-fast Paxos*, contorna tal impossibilidade elegendo um coordenador distinto que é responsável por iniciar novas rodadas, o Líder.

No *Collision-fast Paxos*, o Líder não tem a função de centralizar as propostas no protocolo, como ocorre no *Paxos*; sua função é coordenar os agentes de modo a garantir que o protocolo termine, sendo assim responsável pelas etapas de reconfiguração do sistema em caso de falhas.

Aqui utilizamos o detector provido pelo Akka como base para nossa implementação do eleitor de Líder. Para a eleição, foi usado um simples método determinístico de escolha

do menor identificador entre os *proposers* presentes no sistema (de acordo com o módulo *membership manager*). Isso é possível pelo fato dos nós já possuírem um identificador único.

Uma vez selecionado o Líder, este *proposer* permanece com está função enquanto não houver suspeita de que está falho pelo detector de falhas. Embora esta eleição não atenda o critério de estabilidade do Ω, na prática, ela se mostrou boa o suficiente para a execução do protocolo de difusão atômica.

O Líder é o único agente que pode iniciar uma nova rodada, o que acontece na primeira vez que o protocolo é executado ou na ocorrência de alguma falha. Uma rodada é uma tentativa de terminar uma instância e são identificadas por uma tupla na forma $\langle n, c, cf \rangle$, onde n é um número natural (inicialmente 0), c é o coordenador dessa rodada e cf é o conjunto ordenado de *Collision-fast proposers* dessa rodada. Tal arranjo possibilita uma ordenação total das rodadas utilizando uma classificação lexicográfica e garante para cada coordenador uma infinidade de rodadas para cada conjunto de *Collision-fast proposer* existente.

A rodada inicial de qualquer coordenador c, é dada por: $\langle 0, c, cf \rangle$, assim, para qualquer par de coordenadores (C_0 e C_1) e subconjunto de *Collision-fast proposers CF* pertencente ao conjunto de *Collision-fast proposers* possíveis, as respectivas rodadas iniciais de cada coordenador seria: $\langle 0, C_0, CF_0 \rangle$ e $\langle 0, C_1, CF_1 \rangle$, onde CF_0 não é obrigatoriamente diferente de CF_1.

Como cada coordenador possui um identificador único, temos que $C_0 \neq C_1$, supondo que seja possível ordenar os identificadores dos coordenadores (e.g. $C_0 \leq C_1$), intuitivamente, temos que C_0 possui infinitas rodadas distintas incrementando n para cada subconjunto CF_0 existente, e o mesmo se aplica a C_1.

Em caso de falha de algum *Collision-fast proposer*, o líder pode selecionar um novo conjunto de *Collision-fast proposers* para a nova rodada, assim os *proposers* falhos podem ser excluídos desse conjunto, possibilitando que o protocolo volte a se comportar como *Collision-fast* e garantindo o progresso do algoritmo.

O Líder foi implementado como uma máquina de estado que possui apenas um estado contendo o identificador do líder atual, inicialmente nulo. Quando notificado pelo *membership manager* o oráculo de eleição de líder escolhe o novo líder, usando a função determinística citada acima, atualiza seu estado e envia uma mensagem aos demais agentes notificando a mudança. Algum *proposer* que venha a receber essa mensagem (que pode ser uma mensagem interna apenas) e verificar que é o novo líder, inicia uma etapa de reconfiguração do sistema, executando a *Phase1* do protocolo descrito em (SCHMIDT; CAMARGOS; PEDONE, 2007).

Caso o Líder venha a falhar, um novo é eleito e uma nova etapa de configuração é iniciada. Isso pode levar a um atraso na tomada de decisão do protocolo, sendo necessário mais de dois passos de comunicação para se alcançar uma decisão em caso de falhas.

Contudo, na ausência de falhas e suspeitas indevidas, o sistema se estabiliza e passa a entregar mensagens em dois passos de comunicação.

3.1.2.4 *Collision-fast Oracle*

Como especificado em (SCHMIDT; CAMARGOS; PEDONE, 2007), os únicos *proposers* que são permitidos propor e ter seus valores aprendidos em dois passos de comunicação são aqueles que desempenham o papel de *Collision-fast proposers* para alguma rodada no protocolo. O conjunto de *Collision-fast proposers* é escolhido durante a etapa de configuração do protocolo, *Phase1*, e permanece inalterado até que o detector de falhas suspeite de algum problema e, nesse caso, um novo conjunto pode ser escolhido.

Esse mecanismo foi implementado na nossa *framework* na forma de um oráculo de seleção de *Collision-fast proposers*, personificado por um ator existente em cada réplica, e responsável por escolher o conjunto de *Collision-fast proposers* dado um conjunto inicial de *proposers*.

Esse oráculo é requisitado durante a etapa de configuração do protocolo (*Phase1*) e a cada etapa de reconfiguração que venha a acontecer decorrente de alguma falha que afete o conjunto previamente escolhido. Assim, o protocolo consegue voltar a ser *Collision-fast* após se recuperar de uma falha, retornando automaticamente ao estado normal do sistema, ou seja, se auto-estabilizando.

Quando o coordenador de alguma rodada do *Collision-fast Paxos* deseja reconfigurar o conjunto de *Collision-fast proposers* (e.g quando ocorre alguma falha), ele solicita ao seu oráculo que escolha tal conjunto com base nas informações que possui, por meio do módulo de detecção de falhas, sobre os *proposers* "saudáveis".

Ao receber uma mensagem *ChooseCFPSet* um oráculo escolhe o novo conjunto de *Collision-fast proposers* e notifica o coordenador da rodada através da mensagem *CFPSet*, como descrito no Algoritmo 3.1. Por meio dessa mensagem, um coordenador c mantém atualizado um conjunto de *proposers* (*activep[c]*) que acredita ser "saudáveis" e que poderá garantir progresso, como descrito na Seção 2.8.

Para escolha do conjunto, foi implementado um método de seleção aleatória com um limite de tamanho do conjunto de *Collision-fast Proposers* (*randomChoice(P, n, S)*), em que basicamente *proposers* são inseridos em um conjunto S aleatoriamente, desconsiderando os suspeitos de falhas (i.e apenas o conjunto de *proposers* "saudáveis" P é considerado), até que um limite de tamanho n seja alcançado.

Futuramente pretende-se modificar o algoritmo aleatório para um que utilize melhor as heurísticas geradas pelo detector de falhas na escolha do novo conjunto de *Collision-fast proposers*, possibilitando assim que tal conjunto reflita a visão do detector de falhas no sistema, selecionando aqueles *proposers* que apresentam menor histórico de falhas, e não apenas exclua os *proposers* suspeitos.

Algoritmo 3.1 Collision-fast Proposer Oracle

Replicas: current set of replicas.
$C(i)$: round i's coordinator.
Chosen: set of proposers currently chosen during configuration phase; initially {}.
P: the healthy set of proposers given by failure detector.
n: number of collision-fast proposers to choose.

1: $CFPOracle(rep, r) \triangleq$
2: **pre-conditions:**
3: $rep \in Replicas$
4: received $\langle$"`ChooseCFPSet`", $P, n\rangle$ from $C(r)$
5: **actions:**
6: $Chosen = randomChoice(P, n, \{\})$
7: send $\langle$"`CFPSet`", $Chosen\rangle$ to $C(r)$
8: $randomChoice(P, n, S) \triangleq$
9: **actions:**
10: **if** $\mid P \mid = 0$ **or** $n = 0$ **then** S
11: **else** $cfp \leftarrow Random(P)$ **and** $randomChoice(P \setminus \{cfp\}, n - 1, S \cup \{cfp\})$

Como mostrado na Seção 2.6 e provado em (SCHMIDT; CAMARGOS; PEDONE, 2007), o problema de *M-Consensus* pode ser reduzido ao *Consensus*, logo o algoritmo *Collision-fast Paxos* que resolve *M-Consensus*, pode resolver *Consensus*, ou seja, ser reduzido ao *Paxos* utilizando um conjunto de *Collision-fast proposers* com cardinalidade 1. Nesta redução, esse único *Collision-fast proposer* corresponderia ao Líder, pois é o único que pode propor. Assim, o oráculo de *Collision-fast proposers* que propomos aqui poderia ser utilizado para reconfigurar Paxos com base em observações do ambiente obtidas pelo detector de falhas ou outras políticas aplicadas ao sistema.

3.1.3 TLA+

TLA+ é uma linguagem formal de especificação, desenvolvida por Leslie Lamport e usada para projetar, modelar, documentar e verificar sistemas concorrentes. Por se tratar de uma linguagem formal lógica e matemática, a precisão das especificações escritas nesta linguagem tendem a solucionar muitas falhas de projetos antes de sua implementação. Isso também possibilita que tal especificação possa ser verificada por um verificador de modelo (do inglês *model checker*) permitindo encontrar violações das propriedades de invariância do sistema, como *safety* e *liveness*.

Desta forma, sua transcrição para código utilizando uma linguagem funcional, é de certa forma direta. Porém as especificações tendem a se manter concisas e normalmente assumem algumas pré-condições que podem demandar um trabalho não esperado durante a implementação do sistema especificado, como por exemplo, que se conheça de ante-mão todos os agentes participantes do protocolo, cenário esse que na prática exige uma etapa de descoberta ou pré-configuração.

Outro exemplo comum de dificuldade de tradução especificação/código tem a ver com

como a comunicação é representada em variantes do *Paxos*. Nestas especificações, feitas em TLA+, há um conjunto de mensagens que pode ser acrescido por qualquer processo que queria enviar uma mensagem e subtraído por qualquer um que queira receber uma mensagem; não há noção implícita de falhas de transmissão ou ordem na entrega.

O protocolo *Collision-fast Atomic Broadcast* foi especificado em TLA+, e sua implementação teve como base tal especificação. O código foi implementado de forma a manter certa proximidade com a especificação, no intuito de aumentar a confiança de corretude da implementação. Não é objetivo desse trabalho, contudo, construir um mecanismo de tradução de especificação para código.

Em 3.2 é mostrado o código referente a *Phase2Prepare* do protocolo, cuja à especificação em TLA+ está a direita. Como pode ser observado o código não é uma tradução direta da especificação, porém se aproxima muito da brevidade das ações especificadas, o que proporciona uma maior legibilidade e confiabilidade da corretude da implementação para os desenvolvedores, não descartando outros métodos de verificação e teste.

Na *Phase2Prepare* do protocolo um *proposer* ao receber uma mensagem do tipo $2S$, enviada pelo coordenador da rodada atual para todos os *proposers* e/ou *acceptors*, verifica se está em uma rodada menor que a que veio na mensagem do coordenador, e caso afirmativo atualiza sua rodada e seu valor para o recebido na mensagem, valor este que pode ser do tipo especial *Bottom* ou um *v-mapping* completo, como descrito na *Phase2Start* do protocolo (SCHMIDT; CAMARGOS; PEDONE, 2014).

Algoritmo 3.2 – Código da Phase2Prepare

```
1  def phase2Prepare(msg: Msg2S,
2          state: ProposerState):
3  ProposerState = {
4    if(prnd < msg.rnd) {
5      checkAndUpdateRound(msg.rnd)
6      if(msg.value.isEmpty) { //isBottom
7        return state.copy(pval = None)
8      }
9      return state.copy(pval = msg.value)
10   }
11   state
12 }
```

MODULE *DistCFPaxosLiv*

$$
\begin{array}{l}
Phase2Prepare(p, r) \triangleq \\
\land\ prnd[p] \prec r \\
\land\ \exists\, v \in ValMap : \\
\qquad \land\ [type \mapsto \text{“2S”},\ rnd \mapsto r,\ val \mapsto v] \in msgs \\
\qquad \land\ \lor\ \land\ v = Bottom \\
\qquad\qquad\qquad \land\ pval' = [pval \ \text{EXCEPT}\ ![p] = none] \\
\qquad\qquad \lor\ \land\ v \neq Bottom \\
\qquad\qquad\qquad \land\ pval' = [pval \ \text{EXCEPT}\ ![p] = v[p]] \\
\land\ prnd' = [prnd \ \text{EXCEPT}\ ![p] = r] \\
\land\ \text{UNCHANGED}\ \langle aVars,\ cVars,\ oVars \rangle
\end{array}
$$

O uso de especificação em TLA+ no desenvolvimento de protocolos distribuídos, facilita o entendimento e identificação de possíveis problemas e adaptações durante a etapa de projeto do sistema, melhorando a qualidade do protótipo.

3.1.4 *Collision-fast Atomic Broadcast*

Como descrito anteriormente, os agentes do protocolo são implementados como atores do Akka e desempenham funções bem específicas. Operam como máquinas de estado, alterando seu estado interno ou transitando por outros estados a cada mensagem recebida. No nosso modelo cada agente, ao receber uma mensagem cria *Futures* que encapsulam

seu estado atual e a operação que deve ser executada.

Após a execução dos *Futures*, que pode ser encadeada, o resultado pode ser acessado de maneira síncrona ou assíncrona. Na nossa implementação todas as ações são assíncronas, exceto durante a etapa de configuração, que utiliza de um *Singleton* para sincronizar a configuração de todas as réplicas.

3.1.4.1 *Futures* e *Promises*

Futures fornecem um meio para realização de operações paralelas de forma eficiente e não bloqueante. Um *Future* é um objeto com um atributo que pode, ou não, possuir um valor naquele momento, ou seja, o atributo pode vir a possuir um valor num futuro. Geralmente esse valor é obtido de operações concorrentes, e pode ser utilizado em seguida por outras operações mesmo que as anteriores ainda não tenham acabado. Compor tarefas concorrentes dessa forma, tende a resultar em um código mais rápido, assíncrono, não bloqueante e paralelo, pois não há necessidade de se esperar por uma resposta para se executar outras operações.

Assim, de forma resumida, *Futures* e *Promises* podem ser entendidos como um objeto que atua como um substituto para o resultado de uma operação que será executada, e que inicialmente tem um valor desconhecido. Em situações onde uma operação demandaria certo tempo de processamento e impediria o processo executor de continuar realizando outras operações até que a primeira termine, pode-se utilizar de *Futures* para fornecer um resultado temporário de modo a não bloquear o processamento, atribuindo um resultado futuro, e portanto, permitindo que as operações sejam realizadas em paralelo.

Em Scala, *Futures* são considerados variáveis que permitem apenas operações de leitura, enquanto que *Promises* são variáveis que podem ser atribuídas uma única vez e definem um resultado para um *Future*. Resolver um futuro é atribuir um valor a um dado *Future*.

Nossa implementação faz uso de *Futures* e *Promises* no tratamento das mensagens recebidas pelos agentes, que são atores do Akka. Cada requisição recebida engatilha uma função que recupera o estado atual do ator (que é um *Future*) e cria uma promessa a despeito do estado resultante do processamento dessa mensagem.

Essa função retorna um estado futuro para o ator, que será resolvido em algum instante pelo *system dispatcher* do Akka, não havendo necessidade de aguardar o resultado do processamento. Desta forma, o ator fica livre para tratar novas requisições. Logo, várias mensagens podem ser tratadas paralelamente e seus estados (futuros) encadeados, compondo uma cadeia de transições para estados futuros da réplica.

Na Figura 12 é mostrada a relação entre o *system dispatcher* do nosso sistema de atores e os agentes do protocolo. O *system dispatcher* nada mais é do que um contexto de execução necessário para executar os *Futures*(e.g. *Thread Pool*).

Os estados de cada agente são representados pelas cores: verde, amarelo e laranja, uma para cada tipo de agente. Cada *Future* criado (seta azul) é visto como uma tarefa a ser executada pela *Thread Pool*. As alterações no estado do agente, fruto do resultado das execuções dos *Futures* são representadas pelos círculos, para cada estado.

Assim, essas alterações são entregues aos agentes, completando os *Futures* criados (seta vermelha), e podem gerar novos estados para cada agente, o que consequentemente, pode levar a uma alteração de comportamento do agente a cada mensagem recebida, tornando assim o sistema dinâmico e sensível a alterações em tempo de execução.

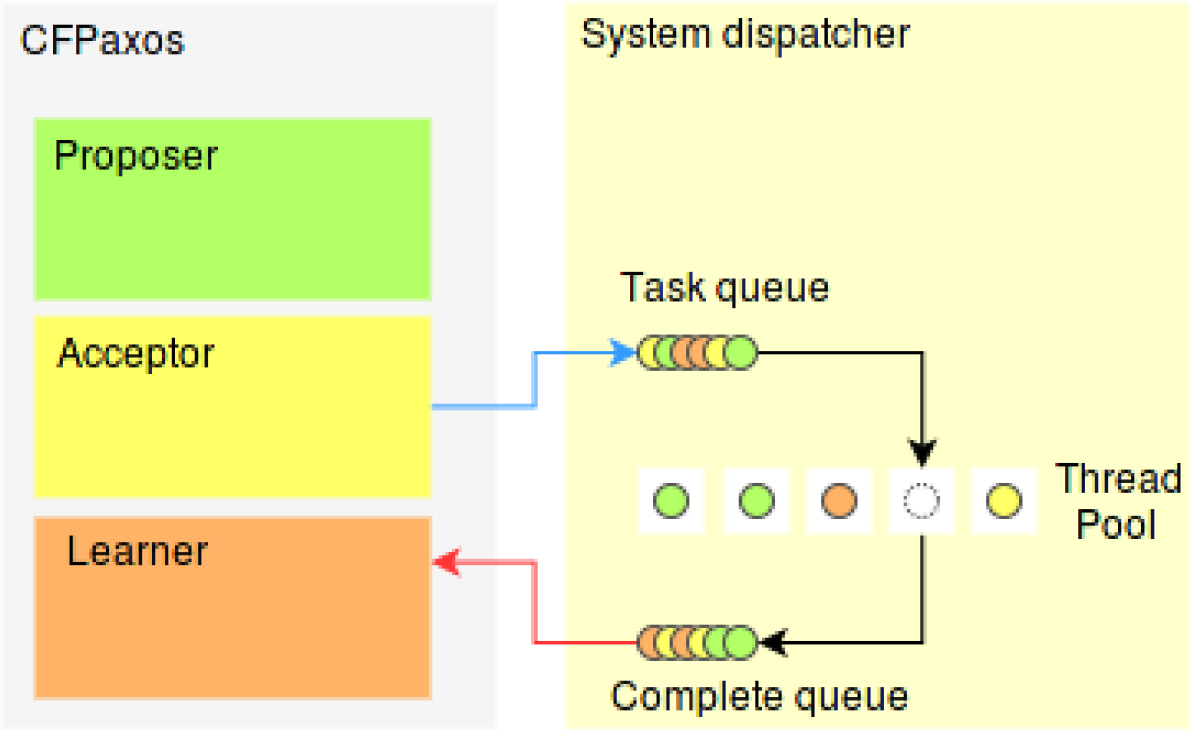


Figura 12 – Relação entre o módulo *Collision-fast Paxos* e o despachante de processos do sistema em uma réplica.

3.1.4.2 *Async*

O uso de *Futures* exige funções de "rechamadas" (*callbacks*) que são executadas quando um futuro é completado, sendo este resolvido ou tendo falhado. Em casos onde os resultados de uma operação dependem de outras, é comum encadear várias instâncias de *Futures*. Isso pode levar a situações em que o gerenciamento de inúmeras funções de *callback* se torna complexo e pode acabar prejudicando o desempenho proporcionado pelo uso de *Futures*, tal cenário é conhecido como *callback hell*. Além disso, tais construções dificultam a leitura, manutenção e entendimento do código.

Existem algumas soluções para o problema citado anteriormente, uma delas foi utilizada nessa trabalho por meio de uma biblioteca *Async*[2], que basicamente disponibiliza

[2] <https://github.com/scala/async>

macros para execução de tarefas assíncronas utilizando *Futures* e *Promises* mas que esconde toda a complexidade gerada pelo encadeamento de *callbacks*.

Desta forma, torna-se possível utilizar *Futures* sem perder legibilidade e coesão do código. Os trechos de código mostrados em 3.3 e 3.4 demonstram tal fato. Ambos os códigos são de uma mesma ação, a realizada por um *collision-fast proposer* que ainda não propôs nessa instância, ao receber uma mensagem do tipo $2A$ de outro, informando que algum valor diferente de *Nil* já foi proposto. Nesse caso, ele atualiza o valor do seu estado e envia *Nil* diretamente para os *learners*, na tentativa de tornar o *v-mapping* aprendido completo, ou seja, forçar o encerramento dessa instância (SCHMIDT; CAMARGOS; PEDONE, 2007).

Como mostrado no Algoritmo 3.4 é utilizada de uma chamada *await*, na linha 6, para esperar de forma assíncrona o estado anterior do ator (*Future*), para assim verificá-lo e utilizá-lo na computação atual.

Essa "espera" não é síncrona, ou seja, o ator não fica bloqueado esperando o resultado, ao invés disso, ele usa uma *Thread* do *System Dispatcher* para esperar por esse resultado. Isso ocorre apenas quando há o processamento de mensagens para uma mesma instância que alteram um estado, ou seja, quando há a possibilidade de ocorrer conflito para operações em um mesmo estado para esse ator.

Por exemplo, suponha que um *Collision-fast proposer* comece a executar a fase *propose*, como descrita no protocolo (SCHMIDT; CAMARGOS; PEDONE, 2014). Nessa fase, após receber uma mensagem de algum cliente, o *Collision-fast proposer* altera seu estado, mudando seu *pval* para o valor que irá propor (i.e. recebido pelo cliente), e propõe tal valor, enviando uma mensagem *Propose* para todos os *acceptors* e demais *Collision-fast proposers* daquela rodada.

Já que o estado é um *Future*, ou seja, o estado pode ainda não ter sido alterado. Esse estado *futuro* criado pela mensagem *Propose* é retornado pela função, liberando o ator para processar mais mensagens até que essa finalize.

Nesse meio tempo, suponha que esse *proposer* receba um mensagem $2A$ de outro, então ele tenta recuperar seu estado *futuro*, que ocorre com exito caso o *Future* já tenha sido finalizado, caso contrário, esse *Future* da mensagem *Propose* não resolvido, é encadeado com o estado atual da mensagem $2A$, escalonando outra *Thread* para esperar pelos resultados da mensagem *Propose* e depois $2A$, liberando o ator para continuar processando outras mensagens.

Algoritmo 3.3 – Código da *Phase2A* usando *Akka Futures*

```
1  def phase2A(msg: Msg2A,
2              state: Future[ProposerState],
3              config: ClusterConfiguration) (implicit ec: ExecutionContext):
4              Future[ProposerState] = {
5
6    val newState = Promise[ProposerState]()
7    state onComplete {
```

```
8       case Success(s) =>
9         if (isCFProposerOf(msg.rnd) && prnd == msg.rnd && s.pval == None
10          && msg.value.getOrElse(msg.senderId, Nil) != Nil) {
11
12          val nil = Some(VMap[AgentId, Values](id -> Nil))
13          (config.learners).foreach(_ ! Msg2A(msg.instance, msg.rnd, nil))
14          newState.success(s.copy(pval = nil))
15        } else {
16          newState.success(s)
17        }
18      case Failure(ex) =>
19        log.error("2A Promise execution failed, because of a {}\n", ex.getMessage)
20    }
21    return newState.future
22  }
```

Algoritmo 3.4 – Código da *Phase2A* utilizando macros *Async*

```
1   def phase2A(msg: Msg2A,
2               state: Future[ProposerState],
3               config: ClusterConfiguration) (implicit ec: ExecutionContext):
4               Future[ProposerState] = async {
5
6     val oldState = await(state)
7     if (isCFProposerOf(msg.rnd) && prnd == msg.rnd && oldState.pval == None
8       && msg.value.getOrElse(msg.senderId, Nil) != Nil) {
9
10        val nil = Some(VMap[AgentId, Values](id -> Nil))
11        val newState = oldState.copy(pval = nil)
12        (config.learners.values).foreach(_ ! Msg2A(id, msg.instance, msg.rnd, nil))
13        return newState
14    }
15    return oldState
16  }
```

3.1.4.3 Valor

No problema de *M-Consensus*, como descrito na Seção 2.6, agentes devem concordar em um crescente mapeamento de *proposers* para algum valor proposto ou um valor especial *Nil* (SCHMIDT; CAMARGOS; PEDONE, 2007). Esse valor foi implementado como um tipo abstrato em Scala. Assim como em Java, Scala possui *Classes*, mas estas, além de possuírem campos e métodos em seus membros, podem também ter tipos. E assim como métodos podem ser abstratos em Java, métodos, campos e tipos podem ser abstratos em Scala.

Além de membros abstratos, Scala possibilita o uso de parametrização, como em Java, porém na prática, o uso excessivo de parametrização em diferentes classes, ou seja, o aumento do número de "coisas desconhecidas" na implementação, pode levar a um crescimento quadrático do código, como mostrado em (BRUCE; ODERSKY; WADLER, 1997). Assim, o uso de membros abstratos possibilita a generalização de tipos ao tempo que colabora para um código mais enxuto e coeso.

Um *v-mapping* pode ser entendido como uma função, que mapeia um elemento do domínio (*proposers*) para um elemento da imagem (conjunto de valores possíveis ou *Nil*). Todas as operações dos mapas de valores descritas em (SCHMIDT; CAMARGOS; PEDONE, 2007) foram implementadas no tipo imutável **VMap**, que estende a coleção **Map** existente em Scala, que nada mais é do que a função de mapeamento citada anteriormente, que mapeia chaves de um tipo A para valores de um tipo B.

Assim, um **VMap** é uma extensão da coleção **Map** existente por padrão em Scala, que mapeia *AgentId* para *Values*, onde *Values* pode ser algum valor ou *Nil* (que é representado por um valor com o campo Nulo) e um **VMap** é *Bottom* se ele é um **Map** vazio.

3.1.4.4 Intervalo de instâncias

No *Collision-fast Paxos* todas as instâncias de *M-Consensus* compartilham um mesmo coordenador, informação esta que é obtida em cada nó através do módulo de eleição de líder. Isso nos permite manter todas as instâncias sincronizadas com relação à rodada atual (criada pelo líder durante a *Phase1* do protocolo) em todos os agentes. Assim, quando o coordenador executa a *Phase1* do protocolo, esta é executada para todas as instâncias apenas uma única vez. Outra rodada só será iniciada no caso do coordenador ser suspeito de falha, indevidamente ou não, e outro agente se julgar coordenador. Se não houverem suspeitas indevidas, então o líder será estável, como em (CHANDRA; GRIESEMER; REDSTONE, 2007), e o protocolo poderá progredir.

Para representar infinitas instâncias, implementamos um intervalo matemático infinito de números inteiros positivos, onde cada inteiro representa uma instância, sendo a instância inicial a zero. Assim como na matemática, o intervalo denotado por: $[a, b]$ é um intervalo fechado entre os números a e b, ou seja, incluindo a e b; ao início da execução, são pré-configuradas para um coordenador inicial todas as instâncias no intervalo $[0, +\infty)$.

A cada instância aprendida um *learner* notifica o *proposer* local existente na mesma réplica e também realiza a entrega do *VMap* aprendido para algum servidor associado a essa réplica. Cada notificação recebida pelo *proposer* é usada na construção de um intervalo que é usado como base para posteriores propostas.

Quando um *collision-fast proposer* deseja propor, ou seja, recebe uma mensagem com um valor a ser decidido de algum cliente, ele deve verificar qual a última instância em que já propôs (um número inteiro) e quais instâncias em que aprendeu algo (um ou mais intervalos). Assim, guardando essas duas informações, esse *proposer* consegue propor em uma nova instância na qual ele ainda não propôs ou aprendeu nada, como especificado no protocolo (SCHMIDT; CAMARGOS; PEDONE, 2007).

Um intervalo é composto por intervalos menores que estão contidos nele. Por exemplo, suponha que o *proposer* tenha recebido de algum *learner* dois intervalos: $A = [0, 0]$ e $B = [2, 4]$. Na construção de seu intervalo de instâncias já aprendidas, a união dos intervalos resulta em um intervalo disjunto: $A \cup B = [[0, 0], [2, 4]]$. Se posteriormente ele

receber algum intervalo contendo a decisão sobre a instância 1, por exemplo $C = [1, 3]$, o intervalo atual do *proposer* seria atualizado para: $A \cup B \cup C = [0, 4]$.

O intervalo das instâncias aprendidas é atualizado em dois momentos: quando o *proposer* é líder e recebe uma mensagem do oráculo de eleição de líder para iniciar uma nova rodada, momento em que o líder re-executa a *Phase1* do protocolo, iniciando uma nova instância; e também quando um *proposer* recebe uma mensagem de algum *learner* informando uma instância aprendida. A última instância proposta é atualizada sempre que uma nova proposta é feita.

Pode acontecer cenários onde o intervalo de instâncias possuirá lacunas, causadas por instâncias ainda não decididas (e.g. *futures* ainda não resolvidos). Por exemplo, suponha que 3 valores sejam enviados a um *collision-fast proposer*. Como cada mensagem é processada individualmente (cada mensagem é armazenada em uma *mailbox* do ator que a consome, funcionando como uma fila *First In, First Out*), cada valor acarretará em uma nova instância criada pelo *proposer*. Assim, ao final da execução dessa fase *proposal* a última instância em que ele terá proposto será a 2 (pois a inicial é a 0); suponha que duas dessas instâncias (0 e 2) tenham sido decididas e já notificadas ao *proposer* e que a instância 1 por algum motivo esteja demorando e ainda não tenha expirado o *timeout*. O intervalo de instâncias aprendidas nesse *proposer* será: $[0, 0][2, 2]$.

Essa lacuna será preenchida com a re-proposição do(s) valor(es) proposto(s) para essas instâncias ainda não decididas, conduzindo ao progresso do algoritmo. Além disso, tal abstração de intervalos possibilita um gerenciamento de faixas de intervalos que podem ser mantidas em memória, já que não é necessário manter todo o histórico de instâncias aprendidas em memória, que pode vir a ser demasiadamente grande.

3.1.4.5 Detecção de um *Quorum*

Para aprender de forma consistente, um *learner* precisa aguardar o recebimento de mensagens $2B$ de um *quorum* de *acceptors* e o recebimento de mensagens $2A$ de um subconjunto (possivelmente vazio) de *collision-fast proposers* da rodada. A especificação, contudo, deixa a cargo do implementador decidir como exatamente os quoruns são definidos e quando essa verificação deve ser feita.

Em nossa implementação quóruns são definidos com uma maioria de agentes que desempenham a função de *acceptor* para a rodada em questão. A quantidade de *acceptors* no sistema e, portanto, o tamanho dos quóruns, é configurado na inicialização. Resta então a questão da verificação da contabilização dos votos dos membros destes quóruns. É importante destacar aqui que embora só faça sentido verificar se houve formação de um *quorum* após a chegada de um número mínimo de mensagens, depois deste fato, outros quóruns, diferentes do original, podem ser também formados e devem ser verificados. Por exemplo, suponha a seguinte execução:

- ❏ cf-proposers p_1, p_2 e p_3 propuseram, respectiva e simultaneamente os comandos c_1, c_2 e c_3;
- ❏ acceptor a_1 aceitou $\langle p_1, c_1 \rangle$;
- ❏ acceptor a_2 aceitou $\langle p_2, c_2 \rangle$;
- ❏ acceptor a_3 aceitou $\langle p_3, c_3 \rangle$;
- ❏ acceptor a_1 aceitou $(\langle p_1, c_1 \rangle, \langle p_2, c_2 \rangle)$;
- ❏ acceptor a_2 aceitou $(\langle p_2, c_2 \rangle, \langle p_3, c_3 \rangle)$;
- ❏ acceptor a_3 aceitou $(\langle p_3, c_3 \rangle, \langle p_1, c_1 \rangle)$;

Suponha agora que *learner* l recebeu dos *acceptors* a_1 e a_2 mensagens $2B$ com $\langle p_1, c_1 \rangle$ e $\langle p_2, c_2 \rangle$ como valores aceitos. Sendo um *quorum* um conjunto de dois *acceptors*, o *learner* deve verificar se a condição de aprendizado foi atendida, decidindo por aprender o *VMap* vazio. Se na sequência l recebe $\langle p_3, c_3 \rangle$, então deve reavaliar a condição, continuando a aprender o *VMap* vazio, pois nenhum dos possíveis quóruns, i.e., $\{a_1, a_2\}$, ou $\{a_1, a_3\}$, ou $\{a_2, a_3\}$, aceitou um mesmo mapeamento. Na sequência, após receber $(\langle p_1, c_1 \rangle, \langle p_2, c_2 \rangle)$ de a_1, l aprende $\langle p_2, c_2 \rangle$, pois identifica que a_1 e a_2 concordam em parte; ao receber $(\langle p_3, c_3 \rangle, \langle p_1, c_1 \rangle)$ de a_3, l aprende $(\langle p_1, c_1 \rangle, \langle p_2, c_2 \rangle)$, pois a_1 e a_2 concordam sobre $\langle p_2, c_2 \rangle$ e a_1 e a_3 concordam em $\langle p_1, c_1 \rangle$; finalmente, ao receber $(\langle p_2, c_2 \rangle, \langle p_3, c_3 \rangle)$ de a_2, l aprende $(\langle p_1, c_1 \rangle, \langle p_2, c_2 \rangle, \langle p_3, c_3 \rangle)$.

Apenas reavaliar todos os possíveis quóruns sempre que uma nova mensagem chega, pode ser ineficiente se o número de *proposers* for grande. Por isso, propusemos o uso de uma estrutura de dados construída interativamente, aumentada a cada nova recepção de mensagem $2B$, e que permite a determinação rápida do *VMap* a ser aprendido.

Nossa estrutura é uma árvore em que cada nó corresponde aos votos para um mesmo *single-map*, recebidos pelo *learner*. Assim, para cada instância, existe um potencial *quorum* associado para uma dada rodada dessa instância, que é obtido por meio de uma busca em largura na árvore, verificando qual nó possui votos de *quorum*. O valor aprendido é o *VMap* que agrega todos os nós que satisfizerem tal condição.

De forma mais concreta, um voto é representado como uma tupla constituída de ⟨valor, contador, conjunto de *acceptors*⟩, que indica quantos aceites cada valor obteve, e quais *acceptors* votaram nele, para uma dada instância de *M-Consensus*, como mostrado na Figura 13.

A operação de verificação de formação de um *quorum* ocorre de forma assíncrona, com uma frequência muito alta durante a execução do protocolo, a cada recebimento de mensagens $2A$ e $2B$. Como apenas *Collision-fast proposers* podem propor e apenas um valor pode ser proposto por *Collision-fast proposer* por instância, consequentemente, a

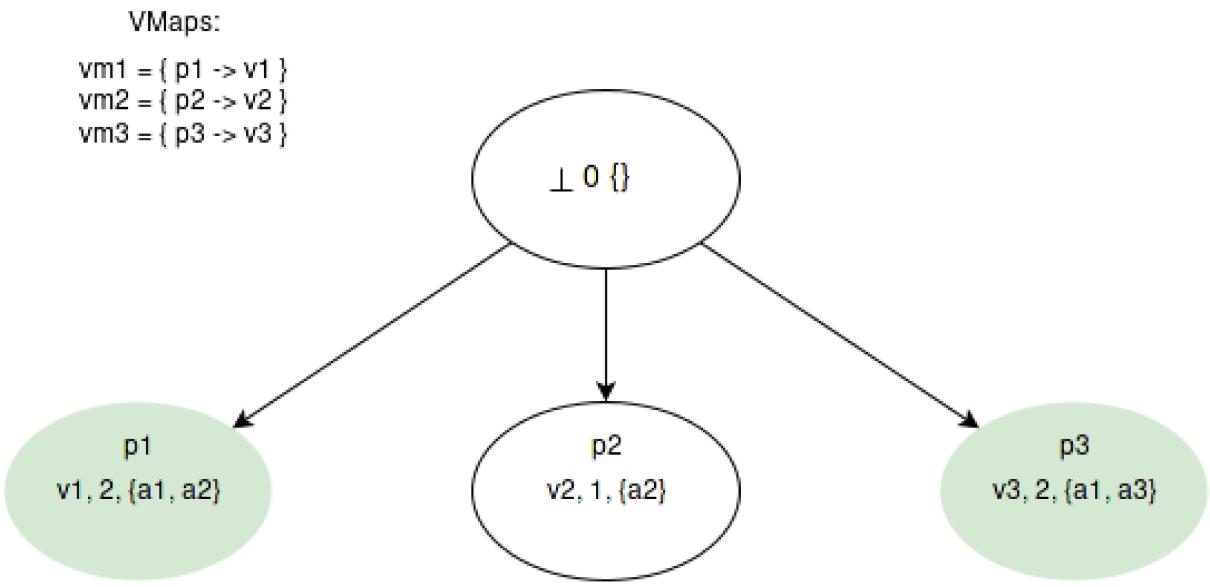


Figura 13 – Exemplo de formação de *quorum*. A raiz representa apenas um estado inicial de cada nó

quantidade de nós na árvore será no máximo a cardinalidade do conjunto de *Collision-fast proposers* existentes para essa rodada, mais um.

Assim, como na ausência de falhas, esse conjunto possui tamanho constante, normalmente configurado na inicialização do protocolo, posteriormente podendo ser modificado apenas durante a etapa de reconfiguração, a árvore possui uma quantidade de nós constantes. Sendo o custo para se verificar a formação do *quorum* igual a $\mathcal{O}(n)$, onde n é o tamanho do conjunto de *Collision-fast proposers*, ou seja, o tamanho de um *v-mapping* completo para cada mensagem recebida.

A Figura 13 ilustra um exemplo de formação de *quorum* considerando uma instância de *M-Consensus* com três *Collision-fast proposers* e três *acceptors*. No exemplo, quatro *single-maps* foram propostos por *Collision-fast proposers* (topo à esquerda). Destas propostas, vm_1 e vm_3 foram aceitas por *quorums*, que o *learner* identifica construindo a árvore apresentada: p_1 possui dois, de três votos possíveis (de a_1 e a_2), para o valor v_1 e p_2 possui dois, de três votos possíveis (de a_1 e a_3), para o valor v_3.

Em nossa implementação desta estrutura, foi utilizada uma versão modificada da estrutura de dados *TrieMap* (também conhecida como *Concurrent Hash Trie*) (PROKOPEC et al., 2012). A *TrieMap* é uma implementação *thread-safe* e livre de *locks* de uma árvore de prefixos cujas chaves são *hashes* mapeados para um vetor (*Hash Array Mapped Trie - HAMT*).

Uma HAMT é uma estrutura de dados formada basicamente pela aplicação de uma função *hash* no conjunto de chaves e o armazenamento dessas chaves em um vetor com base no valor de cada *hash*. Seu uso na nossa implementação é similar ao de um *Map* que possibilita atualizações concorrentes de forma atômica. Tal estrutura foi adaptada para que fosse possível contabilizar a formação de *quorums* para cada instância de *M-*

Consensus.

Uma *TrieMap* possui desempenho constante de operações de busca, devido ao fato do tamanho fixo do *hash* usado como chave, além de utilizar de forma mais eficiente a memória do sistema, pois mantem o mínimo de informação possível nos nós internos. Nessa estrutura, os identificadores dos *Collision-fast proposers* são as chaves e os valores são votos.

3.1.4.6 *Atomic Broadcast*

Apesar de ser relativamente simples resolver Difusão Atômica utilizando *M-Consensus*, alcançar uma solução *Collision-fast* depende do algoritmo de *M-Consensus* utilizado.

Assim como descrito em (SCHMIDT; CAMARGOS; PEDONE, 2007), nossa implementação utiliza infinitas instâncias de *M-Consensus* cada uma identificada unicamente por um número natural para resolução do problema de Difusão Atômica. As instâncias são representadas como um *hash* que mapeia o identificador da instância para um estado futuro do agente, que será resultado de um *future* executado a cada mensagem recebida.

Uma réplica pode atuar como qualquer um dos três tipos de agentes, normalmente possuindo pelo menos um *proposer* e um *learner* para cada instância de *M-Consensus*, podendo ser o mesmo para todas as instâncias de Difusão Atômica.

Para propor, um *proposer p* cria um *Future* contendo o valor que ele deseja propor na menor instância de *M-Consensus i* na qual ele não propôs ou aprendeu nada ainda. Informação essa que é recuperada utilizando os intervalos descritos na Seção 3.1.4.4.

Se uma mensagem de mesma instância for recebida então o processamento desta dependerá do resultado do *Future* disparado pela mensagem anterior. Caso contrário ela pode ser processada paralelamente.

Quando uma decisão é alcançada e o *Future* é resolvido com sucesso, o *learner* atualiza seu estado no mapa de instâncias para o novo valor aprendido, e pode entregar uma sequencia totalmente ordenada para cada decisão de *M-Consensus*, considerando que existe uma ordem total no conjunto de *proposers*.

Assim, como cada réplica controla infinitas instâncias de *M-Consensus* e compartilha de um mesmo coordenador, é possível manter a informação da rodada atual de todos os agentes em todas as instâncias sincronizadas.

3.1.5 Execução especulativa

Primitivas de Difusão atômica garantem que mensagens enviadas para um conjunto de processos são, em algum momento, entregues por todos esses processos em uma mesma ordem total, após a definição desta ordem (DÉFAGO; SCHIPER; URBÁN, 2003), uma abordagem normalmente denominada conservativa.

Na nossa implementação do CFABCAST a entrega pode também ser feita de forma otimista, o que não é definido na especificação do protocolo (SCHMIDT; CAMARGOS; PEDONE, 2007). Assim, ao receber uma mensagem um agente pode entrega-la de imediato, antes do conhecimento da ordem total, permitindo à aplicação processá-la especulativamente. Isso gera um ganho de performance em casos onde a rede é confiável e as mensagens tendem a ser ordenadas naturalmente. Porém, quando a ordem total se tornar conhecida, pode ser necessário reverter a ação especulativa.

Por exemplo, se a mensagem entregue de forma otimista for um comando que divide o valor de X por dois e for executada especulativamente, então se mais tarde chegar uma mensagem que some 17 à X e que deva ser executada antes, então o resultado da divisão estará errado e precisará ser corrigido. Uma forma seria reverter a divisão, somar 17, e então dividir novamente. Outra seria somar 17/2 à X, o que é conhecido como ação compensatória. Caso a execução especulativa não tenha que ser revertida, então a aplicação ganhou tempo, melhorando o tempo de processamento total das mensagens.

O Algoritmo 3.5, possibilita a execução especulativa de um comando, onde o *v-mapping prelearned* é fruto da entrega otimista de mensagens por um *learner* l, via a ação *PreLearn* que é efetuada antes da *Learner*.

Tal algoritmo foi incorporado na implementação do *Collision-fast Atomic Broadcast*, possibilitando a realização de três métodos de entregas:

- ❏ super-otimista: Entrega cada *single map* assim que recebe o primeiro voto (não espera a formação de *quorum*); a verificação ocorre no momento da contagem do voto. Este método possibilita a **execução especulativa** dos comandos.

- ❏ otimista: Entrega um *single map* s se existe um *quorum* de *acceptors* para s (cada valor tem um *quorum*).

- ❏ conservadora: Tipo de entrega padrão, apenas entrega o *v-mapping* quando este é completo (todos os valores pertencentes ao **VMap** possuem um *quorum*), e portanto se tem a posição definitiva de cada comando na sequência gerada pelo **VMap**.

Algoritmo 3.5 Speculative Collision-fast Paxos

prelearned[l]: v-mapping currently pre-learned by learner l during speculative execution; initially $\perp$
1:*PreLearn*(l) $\triangleq$
2: **p**re-conditions:
3: $l \in$ *learners*
4: received $\langle$"2b", $a, r, \langle p, V\rangle\rangle$ **or** $\langle$"2a", $p, r, \langle p, Nil\rangle\rangle$
5: **a**ctions:
6: IF *prelearned*[l][p] $\neq$ *none*
7: THEN **store** *prelearned*[l][p] **and** $v(p)$ for later notification
8: ELSE *prelearned*[l][p] $= v(p)$

O método de entrega *super-otimista* ignora a espera dos *learners* por um *quorum* de *acceptors* para informar a decisão para aplicação, ou seja, assim que um *learner* recebe uma mensagem de um *acceptor*, informando um valor (e.g. comando) aceito ou de um *collision-fast proposer*, informando o valor Nil (o que significa que esse *collision-fast proposer* não irá propor nessa instância e que portanto o *learner* não precisa esperar por um valor dele advindo do *quorum*), esse *learner* pode entregar o valor imediatamente para a aplicação, evitando a espera por decisão pelo *quorum*.

Caso esse valor entregue seja um comando a ser executado no lado da aplicação, e a aplicação suporte execução especulativa, esta pode executá-lo, mesmo o conjunto e ordem total de mensagens não tendo sido ainda definidos pelo *M-Consensus*.

Tal processo acelera a entrega de uma resposta para a aplicação ao passo que sacrifica a consistência, podendo levar a aplicação final a um estado inconsistente devido a execução desordenada dos comandos especulados. Essa inconsistência é resolvida com uma reversão das ações realizadas pela aplicação, que pode ser feita por meio de uma notificação do *learner*, após confirmação do *v-mapping* decidido por um *quorum* contendo a ordem correta de execução dos comandos.

Em casos onde o progresso do protocolo pode ser comprometido, o método de execução especulativa também pode ser utilizado, propiciando uma forma da aplicação final continuar sua execução, porém não garantindo consistência, até que o protocolo se recupere da falha e volte a sua operação normal.

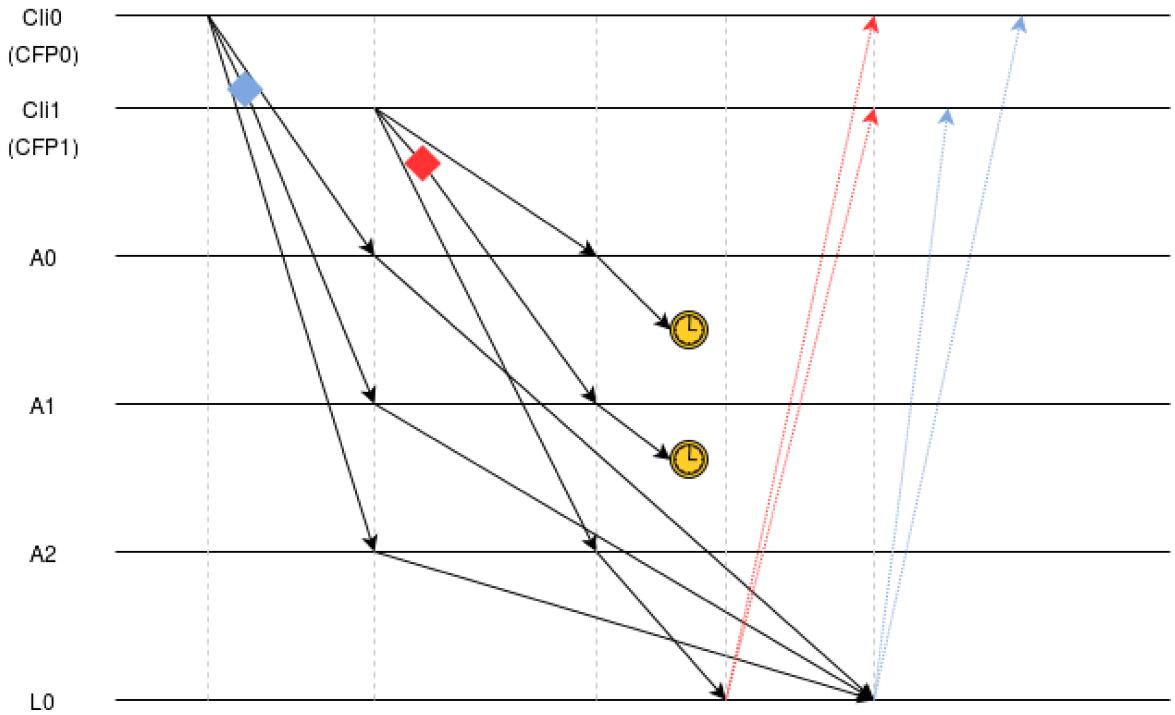


Figura 14 – Exemplo de execução especulativa em que uma inconsistência é detectada, devido a desordenação das mensagens.

Na Figura 14 ilustramos um cenário com dois clientes que são réplicas de um sistema,

e que devido a um atraso na transmissão das mensagens, o uso de execução especulativa leva a uma ordem de execução temporariamente diferente da acordada após resolução do consenso. Como a ordem dos comandos é dada pela ordem de entrega dos valores, antes da formação de um *quorum* para ele, esse método de entrega não assegura consistência forte.

Assim, no exemplo temos que dois comandos diferentes foram propostos pelos dois clientes (Cli_0 e Cli_1), representados pelos losangos (azul e vermelho), cada um por um *Collision-fast proposer* distinto. Supondo a execução especulativa do protocolo, onde o comando vermelho é recebido primeiro pelo *learner* L_0 e especulativamente entregue para aplicação, permitindo que esta já o execute, antes da formação de um *quorum* para esse valor.

Posteriormente esse mesmo *learner* fica sabendo da existência do comando azul, também o entregando para aplicação, e também permitindo a sua execução, resultando em uma ordem de execução de comandos na aplicação (vermelho-azul) antes de um *quorum* ser formado para o comando vermelho.

Considerando que a ordem total entre os comandos é dada pelos identificadores dos *Collision-fast proposers* tal que $CFP_0 \leq CFP_1$, e que a ordem em que tais comandos são executados afeta no resultado da operação na aplicação, será necessário um *rollback* dos comandos quando o *M-Consensus* for resolvido, ou seja, quando o *learner* receber de um *quorum* de *acceptors* o aceite para o *v-mapping*: $\{CFP_0 \rightarrow azul, CFP_1 \rightarrow vermelho\}$ assegurando a ordem total dos comandos, a execução especulativa de vermelho e depois azul precisa ser desfeita para que a consistência seja garantida entre todas as réplicas.

O *Learner* então pode notificar os clientes sobre o problema, informando a ordem correta a ser executada (azul-vermelho). Os clientes por sua vez, desfazem as ação tomadas e as re-executam na ordem correta, informada após resolução do *M-Consensus*, permitindo que o sistema volte a ser consistente.

Capítulo 4

Avaliação

Neste capítulo descreveremos o método de avaliação de nossa implementação, detalhando os ambientes de testes utilizados, *benchmark* e resultados obtidos.

Claramente, a grande vantagem do CFABcast em relação a outros protocolos de difusão atômica é a capacidade de tolerar colisões sem perda de desempenho. Este diferencial se torna mais relevante então em ambientes em que há maior propensão à colisões. Este é o caso em uma topologia comum em *Datacenters*, que considera grupos de nós com baixa latência entre si e alta latência entre nós de outros grupos. Isso porquê mensagens tendem a chegar aos destinatários dentro do mesmo *datacenter* muito mais rapidamente que àqueles de outros *datacenters*, e portanto não são naturalmente ordenadas. Assim, para avaliar o protocolo, foram conduzidos experimentos simulando redes entre *datacenters*.

4.1 Ambiente de Testes

A avaliação dos protocolos de *Atomic Broadcast* foi realizada utilizando o Emulab (EIDE; STOLLER; LEPREAU, 2007), uma infra-estrutura de testes pública disponível para pesquisadores de todo o mundo que oferece uma variedade de ferramentas que auxiliam no desenvolvimento, avaliação e depuração de sistemas. O Emulab possibilita a criação de topologias de redes variadas com uma gama de parâmetros configuráveis, como largura de banda e latência, além de oferecer um ambiente controlado, previsível e repetitivo, permitindo o acesso com total permissão aos nós computacionais e executando o sistema operacional desejado pelo usuário.

O ambiente utilizado no Emulab consiste de dezesseis máquinas (nós computacionais) conectadas através de uma rede *Gigabit Ethernet* em topologia estrela. Cada nó possui um processador *Intel Xeon E5-2630 v3* de 2.40 GHz, 65 Gb de memória principal e operam utilizando o sistema operacional CentOS 7.

Dos dezesseis nós utilizados no Emulab, um foi utilizado apenas para monitoramento e coleta dos dados dos experimentos, como vazão e latência, por meio de uma biblioteca

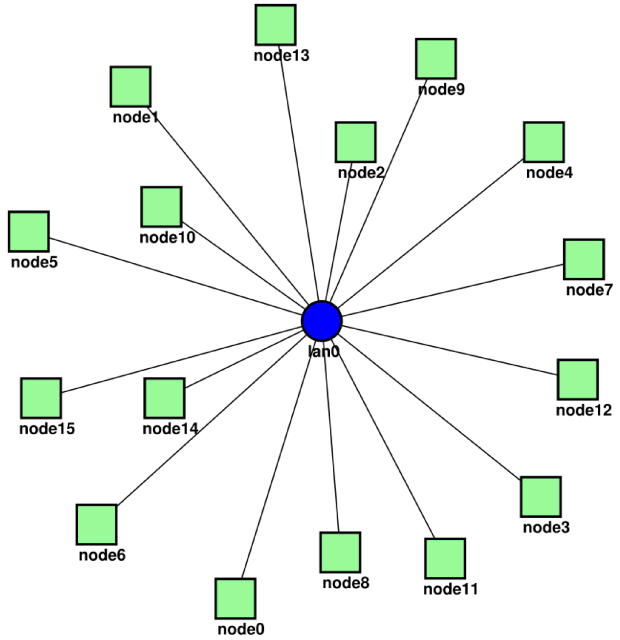


Figura 15 – Topologia estrela utilizada nos testes do Emulab

open source chamada Libmcad [1]. Dez, dos demais nós, foram utilizados como clientes e os cinco restantes como réplicas do protocolo.

Dentre as réplicas uma (R_0) foi selecionada para inserir um atraso na rede de 8 milissegundos, e as demais foram configurados para uma latência de até 2 milissegundos, resultando em uma latência minima de rede de 10 milissegundos em uma comunicação entre duas réplicas "distantes" e 4 milissegundos entre duas "próximas". Assim categorizamos dois grupos de réplicas, rápidas e lentas, aos quais os clientes se conectam, como ilustrado na Figura 16.

Tal atraso pode prejudicar o desempenho de protocolos de acordo que centralizam suas propostas em apenas um agente (e.g R_0), como é o caso do *Paxos* clássico, em que apenas o líder pode ter suas propostas aceitas em dois passos de comunicação. O *Collision-fast Paxos* não possui tal problema, já que mais de um agente pode atuar como *Collision-fast proposer*, ou seja, pode ter suas propostas aceitas com latência igual a do líder no *Paxos*.

4.2 Cenários

Para a realização dos testes foi utilizada a biblioteca Libmcad, que possibilita avaliar o desempenho de diferentes algoritmos de *multicast* atômico. Para o nosso caso, utilizamos a biblioteca em modo *broadcast*, realizando um *multicast* para um grupo apenas. Assim foi criada uma interface para adicionar nossa implementação de replicação de máquinas de estados à Libmcad, estendendo assim a biblioteca para suportar o protocolo *Collision-fast Atomic Broadcast*.

[1] https://bitbucket.org/kdubezerra/libmcad

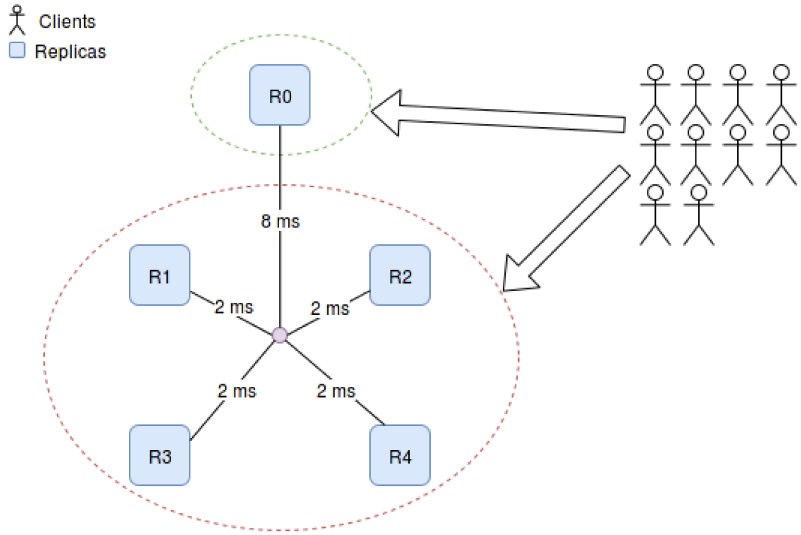


Figura 16 – Divisão lógica entre os nós (réplicas e clientes) considerando a latência de comunicação entre eles.

O desempenho do protocolo *Collision-fast Paxos* foi avaliado utilizando nosso *framework* de replicação de máquinas de estado apresentado no Capitulo 3, tendo como objeto de comparação o protocolo *Paxos* em sua forma clássica (LAMPORT, 2001) e uma implementação do *Multi Ring Paxos* disponível na Libmcad.

Os experimentos consistem basicamente de clientes que enviam uma mensagem, contendo um número de sequência e um identificador único, para alguma réplica do sistema. Essa réplica realiza um *broadcast* dessa mensagem que, em algum momento, é proposta na camada de *consensus* do nosso *framework*, utilizando o *Paxos*, o *Collision-fast Paxos* ou o *Multi Ring Paxos* para atingir um consenso.

Quando algo é aprendido por algum *learner* na camada de *consensus*, este notifica a réplica ao qual ele está associado que por sua vez entrega para o cliente o que foi aprendido. Assim, a aplicação simplesmente retorna uma mensagem de confirmação para o cliente informando o número de sequência que foi decidido, possibilitando o cálculo da latência e vazão de cada protocolo.

Nos experimentos foram utilizadas três e cinco réplicas, cada uma delas contendo um agente responsável por cada função do protocolo. Os clientes são criados dinamicamente a cada caso de teste de forma progressiva, sendo criados um total de 10 clientes concorrentes, que se conectam às réplicas utilizando uma política *Round Robin*.

Em ambos os cenários, com três e cinco réplicas, o *quorum* estipulado representa uma maioria de agentes desempenhando a função de *acceptor* na réplica, ou seja, para três réplicas, um *quorum* é formado se um *learner* recebe mensagens de pelo menos dois *acceptors* quaisquer. Já para cinco réplicas, três *acceptors* são necessários para formação de um *quorum*.

Os clientes são responsáveis por gerar a carga dos testes, e após todos estarem co-

nectados às réplicas do sistema, cada cliente executa um *loop* enviando uma quantidade de mensagens definidas em uma janela de propostas. Nesse trabalho a janela para os testes foi de dez mensagens simultâneas por cliente, assim, cada cliente envia dez mensagens e aguarda pela resposta de pelo menos uma, para poder enviar a decima primeira, e assim sucessivamente. Por exemplo, caso o cliente receba a confirmação de três mensagens aprendidas, ele está permitido a enviar mais três novas propostas, e assim o faz em seguida.

Essa janela de propostas tem o intuito de simular no nosso *framework* um ambiente de propostas concorrentes em que pode haver colisões na camada de consenso, assim como acontece em sistemas reais com alta taxa de concorrência que utilizam consenso para manter consistência entre as réplicas.

Os resultados apresentados na Seção 4.4 contemplam apenas os dados referentes a entrega da decisão ao cliente assim que um valor é decidido. Denominamos tal entrega de otimista no nosso protocolo, pelo fato do *v-mapping* ainda não ser completo no momento da entrega, como descrito na Seção 3.1.5.

4.3 Métricas

Como o *Fast Paxos*, o protoloco *Collision-fast Paxos* foi desenvolvido com o intuito de eliminar um passo de comunicação para se alcançar consenso. Porém, diferente do *Fast Paxos*, o *Collision-fast Paxos* trata colisões de forma eficiente, garantindo a latência ótima de dois passos de comunicação mesmo em presença de colisões e não requer *quorums* maiores que o *Paxos* clássico.

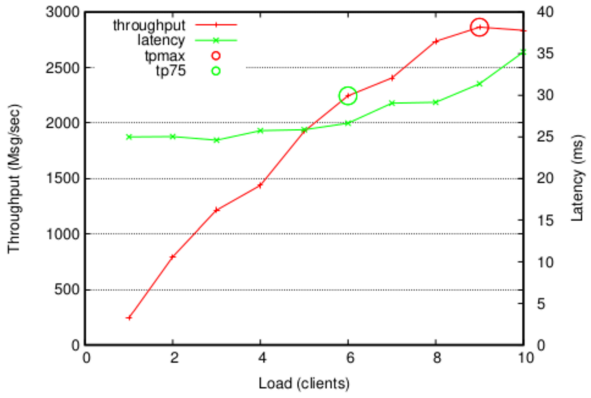
Desta forma o maior benefício de se utilizar o *Collision-fast Paxos* para replicação de máquinas de estado é uma diminuição na latência fim-a-fim em relação ao cliente, ao passo que possibilita aprender mais de um valor por instância, em caso de propostas concorrentes, o que aumenta a vazão do sistema.

Assim, duas métricas principais foram avaliadas em cada protocolo de consenso testado, a latência e a vazão desses protocolos quando usados para replicação de máquinas de estado. A latência fim-a-fim é definida como o tempo decorrido a partir de uma requisição do cliente até o recebimento de uma resposta correspondente. Para mensurar tal latência nos nossos experimentos, monitoramos com o auxilio da *Libmcad*, o tempo de viagem (*roundtrip*) de cada mensagem para cada cliente.

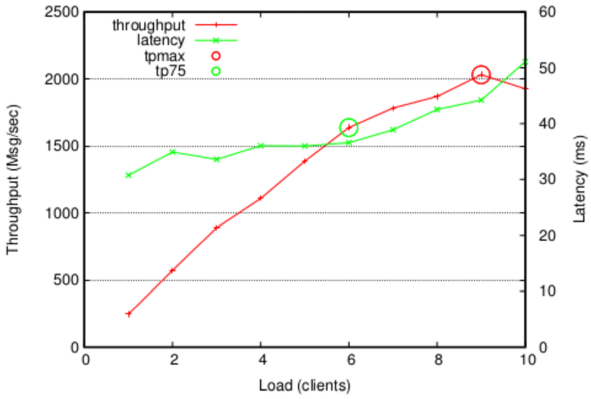
Já a vazão do sistema reflete o desempenho do protocolo sob a perspectiva do número de requisições por segundo que ele consegue tratar. Para a avaliação da vazão são criados até dez clientes concorrentes, que enviam mensagens de forma intermitente de acordo com a janela de propostas previamente configurada. Para os testes foi utilizada uma janela de dez requisições simultâneas por cliente.

4.4 Avaliação dos Resultados

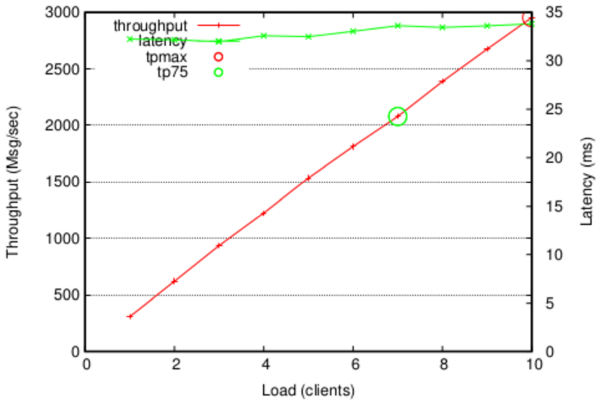
Os resultados apresentados na Figura 17 mostram a latência fim-a-fim (em milissegundos) e a vazão (em número de mensagens por segundo) do sistema replicado utilizando os três protocolos testados para uma carga variável de um a dez clientes concorrentes conectados.



(a) Collision-fast Paxos



(b) Paxos



(c) Multi Ring Paxos

Figura 17 – Latência e vazão dos protocolos por clientes

Como pode ser observado, o protocolo *Collision-fast Atomic Broadcast* mostrou relativa melhor latência nos cenários de testes que os demais protocolos. É possível perceber que o *Collision-fast Paxos* conseguiu manter uma vazão similar ao *Multi Ring Paxos* durante praticamente todo o experimento, e que o *Paxos* como esperado, teve menor vazão, devido ao fato do líder centralizar a proposta de valores. Observe também que o percentil 75 é maior no CFABcast que nos demais protocolos. Contudo, exceto pelo Multi-Ring Paxos, a vazão cai após atingir seu pico.

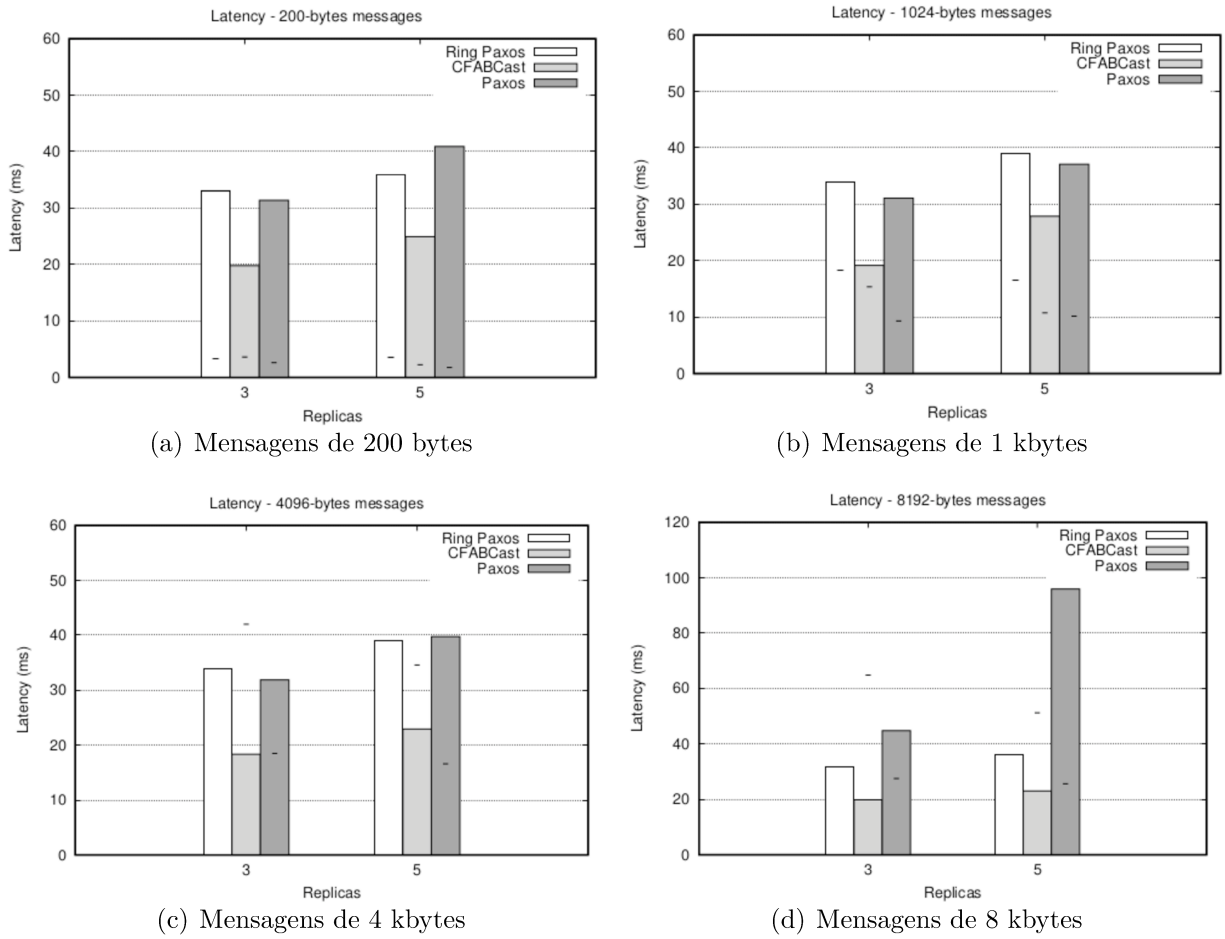
Isso demonstra que o *Collision-fast Paxos*, como suportado pela literatura, de fato consegue aumentar a vazão na resolução do *M-Consensus*, pois mais de um valor pode ser decidido por instância do problema, ao passo que diminui a latência de fim-a-fim observada

pelo cliente, já que consegue manter a latência ótima de dois passos de comunicação mesmo em presença de colisões, como descrito na Seção 2.7.

Porém é preciso ressaltar que a latência mostrada na Figura 17(a) e 17(b) possui significativa variação (mais de 10 milissegundos), diferente da latência mostrada em 17(c), que varia muito pouco no decorrer do tempo. Isso provavelmente é devido a uma ineficiência da nossa implementação desses dois primeiros protocolos do que dos protocolos em si. Trabalhos futuros são necessários para investigar tal comportamento.

4.4.1 Latência

Para demonstrar como os protocolos se comportam para variados tamanhos de mensagens e o impacto disso na latência de fim-a-fim da aplicação, realizamos experimentos alterando o tamanho da mensagem enviada pelo cliente. Para cada cenário de testes, foram utilizadas mensagens com tamanhos de 200 *bytes*, 1024 *bytes*, 4096 *bytes* e 8192 *bytes*. A Figura 18 ilustra tais experimentos com os valores médios de latência para os três protocolos.



(a) Mensagens de 200 bytes

(b) Mensagens de 1 kbytes

(c) Mensagens de 4 kbytes

(d) Mensagens de 8 kbytes

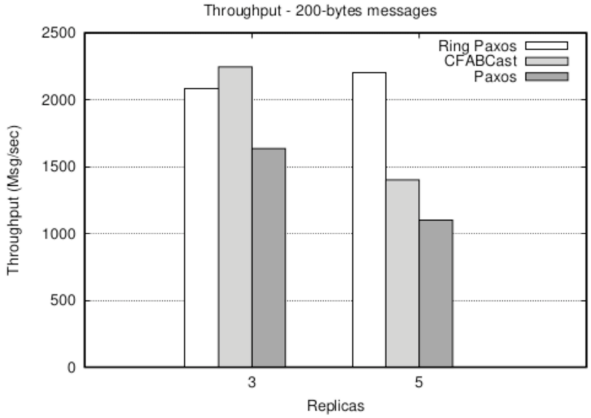
Figura 18 – Latência média dos protocolos variando o tamanho das mensagens, para três e cinco réplicas

É possível perceber que há uma significativa degradação do tempo de resposta do *Paxos* para mensagens de 8 *kbytes*, principalmente no cenário com cinco réplicas mostrado na Figura 18(d). Independente do número de réplicas e do tamanho das mensagens, o *Collision-fast Paxos* manteve uma menor latência média em relação aos demais protocolos.

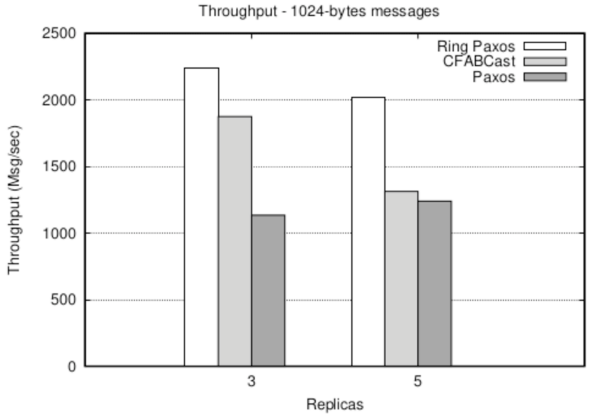
Esse resultado era o esperado, e corrobora com a teoria apresentada em (SCHMIDT; CAMARGOS; PEDONE, 2007), demonstrando que mesmo em presença de colisões, o protocolo *Collision-fast Paxos* possui menor latência que o *Paxos*, devido o uso de propostas rápidas.

4.4.2 Vazão

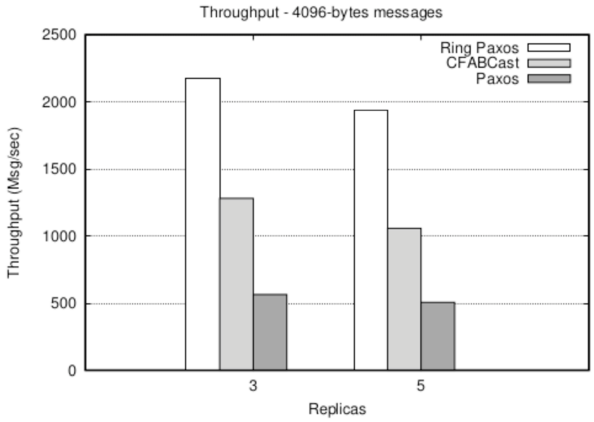
Como apresentado na Figura 19, nos testes o *Collision-fast Paxos* obteve melhor vazão do que o *Paxos* mesmo em presença de colisão, como sugerido em (SCHMIDT; CAMARGOS; PEDONE, 2007). Porém nossa implementação não é performática o suficiente para se equiparar, em todos os casos de testes, ao desempenho do *Multi Ring Paxos*, que foi desenvolvido com o intuito de aumentar a vazão em grupos de comunicação entre processos.



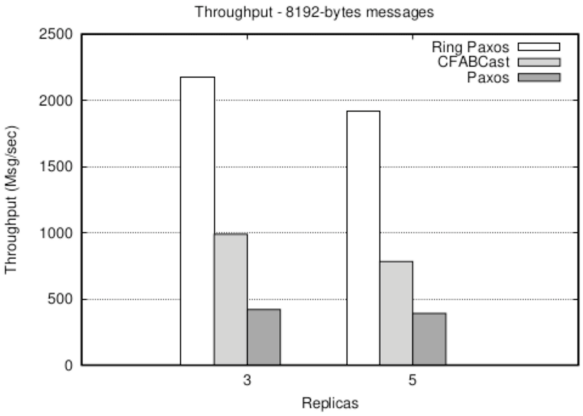
(a) Mensagens de 200 bytes



(b) Mensagens de 1 kbytes



(c) Mensagens de 4 kbytes



(d) Mensagens de 8 kbytes

Figura 19 – Vazão média variando o tamanho das mensagens, para três e cinco réplicas

Capítulo 5

Eventually Collision-Fast Byzantine Atomic Broadcast

Neste capítulo apresentaremos nossa extensão do protocolo proposto em (SCHMIDT; CAMARGOS; PEDONE, 2007), tornando-o tolerante a falhas bizantinas. Demonstraremos como o uso do método de assinatura digital pode ser utilizado para atingir tal objetivo. O algoritmo resultante, contudo, não é mais *collision-fast*, mesmo na ausência completa de falhas, pois em algumas situações precisa esperar um passo de comunicação extra para se certificar de que não está sendo enganado por um *collision-fast proposer* bizantino. Baseado em nosso esforço, conjecturamos que é impossível criar um algoritmo *collision-fast* que tolere falhas bizantinas.

5.1 Assinatura digital

Assinatura digital é um método de autenticação de informação digital, ou seja, uma forma segura de garantir a integridade e procedência de alguma informação transferida digitalmente, geralmente utilizando de criptografia de chaves assimétricas como base para o estabelecimento de uma comunicação confiável.

Basicamente o processo consiste em aplicar uma função *hash* criptográfica (e.g. md5, sha-256) à mensagem e cifrar o resultado utilizando um sistema de chave pública, assinando-a utilizando a chave privada do remetente e enviando o *hash* criptografado junto a mensagem original, tal processo de assinatura está ilustrado na parte esquerda da Figura 20.

Após o recebimento de uma mensagem digitalmente assinada, deve-se verificar sua autenticidade, para isso o destinatário aplica a mesma função *hash* criptográfica na mensagem recebida e compara o resultado com o *hash* original, obtido a partir da decifração da mensagem recebida utilizando da chave pública do remetente. Se os *hashs* forem iguais, a mensagem está íntegra e a assinatura é válida. Este processo é mostrado à direita da Figura 20.

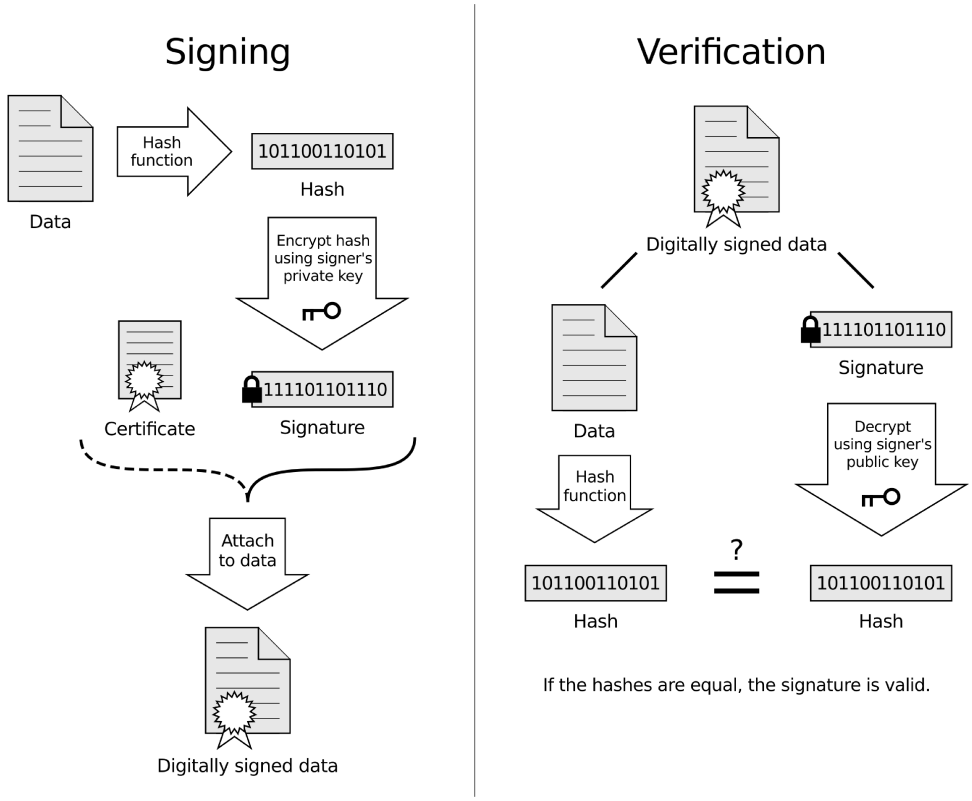


Figura 20 – Etapas do processo de assinatura digital

Fonte: https://en.wikipedia.org/wiki/Electronic_signature

O método de assinatura digital foi utilizado nesse trabalho para possibilitar que o protocolo *Collision-fast atomic broadcast* tolere falhas bizantinas.

5.2 Consenso Bizantino

Em (SCHMIDT; CAMARGOS; PEDONE, 2007) considerou-se um modelo computacional com comunicação por troca de mensagens e assíncrono. Além disso, considerou-se o modelo de falhas *crash-recovery*, isto é, em que agentes podem parar de funcionar por um período arbitrariamente longo, e que segue um protocolo de recuperação ao voltar a executar. Tais processos, contudo, nunca executam ações maliciosas, ou seja, não acontecem falhas Bizantinas.

O problema do Consenso Bizantino diz respeito a resolução do problema de *Consensus* adotando um modelo de falhas semelhante ao citado anteriormente, exceto, que os agentes podem agir de forma maliciosa e arbitrária. Por exemplo, processando incorretamente as mensagens, falsificando sua identidade, corrompendo dados ou se negando a responder alguma requisição, desempenhando assim, um comportamento não esperado no sistema. Tais agentes são comumente denominados Bizantinos (LAMPORT; SHOSTAK; PEASE, 1982) e nosso intuito é permitir que o protocolo CFABCAST tolere as falhas causadas por esses agentes.

Ao contrário do bizantino, um agente correto é aquele que executa um número infinito de tarefas e respeita a especificação do algoritmo que está supostamente executando no sistema. Um agente é considerado correto enquanto não houver suspeita do detector de falhas de alguma ação indevida.

Diversas possíveis soluções para o problema existem. Algumas consideram simplesmente o uso de quóruns maiores, como por exemplo (LAMPORT; SHOSTAK; PEASE, 1982), que apresenta uma solução que considera que o número de agentes falhos (Bizantinos), f, deve ser $f < n/3$ onde n é o número de processos executando o algoritmo. Outra abordagem faz uso de assinatura digital (CASTRO; LISKOV, 1999; RAMASAMY; CACHIN, 2006), que embora simplifique os protocolos em alguns aspectos, pode incorrer em um aumento no tempo de processamento das mensagens e consequente atraso na resolução do consenso.

Em (ABRAHAM; AMIT; DOLEV, 2005) foram provadas condições necessárias e suficientes para se resolver Consenso Bizantino em sistemas assíncronos, de modo que a quantidade necessária de agentes desempenhando a função de *acceptors* deve ser maior ou igual a $3f + 1$, onde f é o número de agentes bizantinos tolerados. Considere o caso de *learners*, por exemplo, que devem aprender baseado nas mensagens dos *acceptors*. De forma simplificada, dado que f *acceptors* poderiam simplesmente parar de executar qualquer instrução, nenhum *learner* pode esperar por mais que $n-f$ aceites sem correr o risco de esperar para sempre. Destes $n-f$ aceites, no máximo f podem ser de processos bizantinos, e portanto $n - 2f$ são de processos corretos. Para que as mensagens dos processo corretos possam se sobrepor às mensagens dos bizantinos, é necessário que $n - 2f > f$, ou seja, $n > 3f$.

Até onde sabemos, não houveram tentativas de se definir algoritmos *collision-fast* (i.e. que possuem uma latência de dois passos de comunicação para múltiplos *proposers* mesmo em presença de colisões) e tolerantes a falhas bizantinos. Como contribuição deste trabalho, tentamos reverter esta situação. O algoritmo proposto aqui, embora *collision-fast* em diversas situações, não o é a despeito de quaisquer falhas. De fato, conjecturamos na Seção 5.4 que um algoritmo *collision-fast* e bizantino não possa existir.

Na nossa proposta utilizamos assinatura digital para auxiliar na resolução do problema de Consenso Bizantino, já que possibilita autenticar os agentes envolvidos nas etapas do protocolo e as mensagens trocadas por eles. Combinadas com um detector de falhas que exclui processos bizantinos, essa abordagem permite que o protocolo se adapte e exclua membros suspeitos de conluio contra o funcionamento correto do protocolo, levando o protocolo a, em algum ponto no tempo, ou *eventually*, se tornar *collision-fast*.

5.3 *Eventually Collision-Fast Byzantine Atomic Broadcast*

Foram propostas modificações no algoritmo apresentado em (SCHMIDT; CAMARGOS; PEDONE, 2007) de forma a garantir um canal de comunicação confiável com unicidade de identificação dos agentes, utilizando o método de assinatura digital. O Algoritmo 5.1 ilustra dois métodos criados para adicionar suporte à assinatura digital no protocolo, considerando uma infra-estrutura de distribuição de chaves confiável, em que basicamente:

1. Todas as mensagens enviadas são assinadas digitalmente (método *sign*) usando métodos de criptografia de chave assimétrica bem conhecidos na literatura, como o RSA .

2. Todas as mensagens recebidas tem a assinatura do remetente verificada (método *verify*), e em caso de incompatibilidade, o erro é reportado ao detector de falhas.

Algoritmo 5.1 Digital signature

Agents: The set of all agents, acceptors, proposers and learners (i.e. $\{A \cup P \cup L\}$).
PKI: Public key infrastructure that manages digital certificates and public-keys.
hash: A cryptographic hash function.
x_{priv}: Private key of a agent x.
x_{pub}: Public key of a agent x.

1: $sign(a, m) \triangleq$
2: **pre-conditions:**
3: $a \in Agents$
4: LET $a_{priv} = a's$ private key
5: **actions:**
6: LET $h = hash(m)$
7: LET $s = encrypt(h, a_{priv})$
8: **return**(m, s)

9: $verify(a, m, s) \triangleq$
10: **pre-conditions:**
11: $a \in Agents$
12: LET $\exists a_{pub}, a_{pub} \in PKI$
13: **actions:**
14: LET $h1 = decrypt(s, a_{pub})$
15: LET $h2 = hash(m)$
16: **if** h1 = h2 **then return** true
17: **else return** false

Para assinar uma mensagem m (*sign*), um agente a que queira transmiti-la, primeiramente, aplica uma função *Hash* criptográfica h nos dados da mensagem m e em seguida criptografa esse *Hash* utilizando sua chave privada a_{priv}. Em seguida, retorna o resultado da encriptação s (assinatura) junto com a mensagem original, para que ambos sejam enviados.

Ao receber alguma mensagem m, um agente b deve verificar se o remetente a é mesmo quem diz ser. Isso é feito consultando a infra-estrutura de chaves públicas *PKI* e verificando se a chave pública de a existe e consegue decriptar a mensagem. Caso afirmativo, o resultado da decriptação é o *hash* da mensagem assinada h_1. Assim, o agente b aplica a mesma função *Hash* na mensagem original h_2 e verifica se os dois *hashes* são iguais, se sim,

o remetente é verdadeiro e a mensagem não foi corrompida, caso contrário, o detector de falhas é notificado sobre uma possível falha de autenticidade ou corrupção de mensagem.

Para distribuição das chaves públicas o uso de uma autoridade certificadora tende a ser o método mais confiável atualmente, apesar de não garantir total segurança da informação e ser um ponto único de falha. Porém, não faz parte do escopo deste trabalho oferecer uma forma confiável de se disseminar as chaves, e pressupõe-se para o correto funcionamento do modelo aqui proposto, que uma infraestrutura de distribuição de chaves não corrompível deva ser utilizada.

Uma vez que cada nó tenha acesso às chaves públicas de todos os demais participantes do protocolo, estes estão aptos a verificarem a autenticidade das informações trocadas, estabelecendo uma canal confiável para comunicação. Como o algoritmo *Collision-Fast Paxos* é estruturado em rodadas e a cada nova rodada o coordenador reconfigura o protocolo, o uso de assinatura digital para troca de informação, possibilita identificar processos maliciosos e excluí-los de rodadas futuras, permitindo uma eventual estabilidade do sistema.

Os Algoritmos 5.2, 5.3 e 5.4 compõem nossa tentativa de possibilitar o uso de assinatura digital no protocolo *Collision-fast Paxos*, em que foram feitas alterações no Algoritmo original 2.1 utilizando os métodos apresentados no Algoritmo 5.1. Todas as mensagens trocadas são assinadas e verificadas digitalmente, garantindo a autenticidade dos envolvidos e segurança da informação trocada.

Algoritmo 5.2 Collision-fast Byzantine Paxos – Primeira Parte

1: $Propose(p, V) \triangleq$
2: **pre-condition:**
3: $p \in Pr$
4: **action:**
5: send (sign (p, $\langle$"`propose`", $V\rangle$)) to $cf \in CF(prnd[p])$
6: $Phase1a(c, r) \triangleq$
7: **pre-conditions:**
8: $c = C(r)$
9: $crnd[c] < r$
10: **actions:**
11: $crnd[c] \leftarrow r$
12: $cval[c] \leftarrow none$
13: send (sign (c, $\langle$"`1a`", $r\rangle$)) to A
14: $Phase1b(a, r) \triangleq$
15: **pre-conditions:**
16: $a \in A$
17: $rnd[a] < r$
18: received (m, s), m = $\langle$"`1a`", $r\rangle$ from $C(r)$ **and** $verify(c, m, s) = true$
19: **actions:**
20: $rnd[a] \leftarrow r$
21: send (sign (a, $\langle$"`1b`", $a, r, vrnd[a], vval[a]\rangle$)) to $C(r)$

Algoritmo 5.3 Collision-fast Byzantine Paxos – Segunda Parte

1: *Phase2Start*(c, r) $\triangleq$
2: **pre-conditions:**
3: $c = C(r)$
4: $crnd[c] = r$
5: $cval[c] = none$
6: $\exists Q : Q$ is a quorum: $\forall a \in Q$ received (m, s), m $= \langle$"1b", $a, r, vrnd, vval\rangle$ **and** *verify*$(a, m, s) = true$
7: **actions:**
8: LET $msgs = [m = \langle$"1b", $a, r, vrnd, vval\rangle$: received m from $a \in Q$])
9: LET $k = Max([vrnd : \langle$"1b", $a, r, vrnd, vval\rangle \in msgs$])
10: LET $S = [vval : \langle$"1b", $a, r, k, vval\rangle \in m$, $vval \neq none]$
11: IF $S = \emptyset$ THEN
12: $cval[c] \leftarrow \perp$
13: send (sign (c, $\langle$"2S", $r, cval[c]\rangle$)) to P
14: ELSE
15: $cval[c] \leftarrow \sqcup S \bullet [\langle p, Nil\rangle : p \in Pr]$
16: send (sign (c, $\langle$"2S", $r, cval[c]\rangle$)) to $P \cup A$
17: *Phase2Prepare*(p, r) $\triangleq$
18: **pre-conditions:**
19: $p \in Pr$
20: $prnd[p] < r$
21: received (m, s), $\langle$"2S", $c, r, v\rangle$ **and** *verify*$(c, m, s) = true$
22: **actions:**
23: $prnd[p] \leftarrow r$
24: IF $v = \perp$ THEN $pval[p] \leftarrow none$ ELSE $pval[p] \leftarrow v(p)$
25: *Phase2a*(p, r, V) $\triangleq$
26: **pre-conditions:**
27: $p \in CF(r)$
28: $prnd[p] = r$
29: $pval[p] = none$
30: **either** $V \neq Nil$ **and** received (m, s), m $= \langle$"propose", $p, V\rangle$ **and** *verify*$(p, m, s) = true$
31: **or** $V = Nil$ **and** received (m, s), m $= \langle$"2a", $r, \langle q, W\rangle\rangle$, $q \in CF(r)$, $W \neq Nil$ **and** *verify*$(q, m, s) = true$
32: **actions:**
33: $pval[p] \leftarrow V$
34: **if** $V \neq Nil$ **then** send (sign(p, $\langle$"2a", $p, r, \langle p, V\rangle\rangle$)) to $A \cup CF(r)$
35: **else** send (sign(p, $\langle$"2a", $p, r, \langle p, V\rangle\rangle$)) to L **and** send (sign(p, $\langle$"2a", $p, r, noop\langle p, < q, W >\rangle\rangle$)) to A

Como ilustrado pelo Algoritmo 5.2, toda mensagem enviada é assinada digitalmente através do método *sign*(a, m), como mostrado na linha 21. E cada mensagem recebida tem seu conteúdo e assinatura verificada pelo método *verify*(a, m, s), onde a é o identificador do agente que enviou a mensagem, m é a mensagem em si e s é a assinatura que foi enviada junto com a mensagem, como mostrado na linha 18.

A verificação retornar um sucesso é uma condição necessária para que o método seja executado. Caso a verificação falhe, o agente que invocou tal método trata o erro gerado e reporta o ocorrido ao Detector de falhas. Caso contrário, o processo continua como

proposto em (SCHMIDT; CAMARGOS; PEDONE, 2007).

O Algoritmo 5.4, representa a etapa de aprendizado do nosso protocolo bizantino, que é uma variação da proposta em (SCHMIDT; CAMARGOS; PEDONE, 2007), na qual um *learner* desconsidera do *v-mapping* aprendido dos *proposers* que falhem na verificação de autenticidade.

Algoritmo 5.4 Collision-fast Byzantine Paxos – Terceira Parte

1: $Phase2b(a, r) \triangleq$
2: LET $Cond1 =$ (received $\langle$"2S", $r, v\rangle$, $a \neq \perp$ **and** $vrnd[a] < r$) **or** $vval[a] = none$
3: LET $Cond2 =$ received $\langle$"2a", $r, \langle p, V\rangle\rangle$, $V \neq Nil$
4: **pre-conditions:**
5: $a \in A$
6: $rnd[a] \leq r$
7: **either** $Cond1$ **or** $Cond2$
8: $\forall m$ that meets ($Cond1$ **or** $Cond2$) **and** ($verify(a, m, s) = true$ **or** $verify(p, m, s) = true$)
9: **actions:**
10: $rnd[a] \leftarrow vrnd[a] \leftarrow r$
11: IF $Cond1$ THEN $vval[a] \leftarrow v$
12: ELSE IF $Cond2$ **and** ($vrnd[a] < r$ **or** $vval[a] = none$)
13: THEN $vval[a] \leftarrow \perp \bullet \langle p, V\rangle \bullet [\langle p, Nil\rangle : p \in Pr \setminus CF(r)]$
14: ELSE $vval[a] \leftarrow vval[a] \bullet \langle p, V\rangle$
15: send (sign(a, $\langle$"2b", $a, r, vval[a]\rangle$)) to L
16: $PreLearn(l) \triangleq$
17: **p**re-conditions:
18: $l \in learners$
19: received m = $\langle$"2b", $a, r, \langle p, V\rangle\rangle$ **or** n = $\langle$"2a", $p, r, \langle p, Nil\rangle\rangle$
20: **either** $verify(a, m, s) = true$ **or** $verify(p, n, s) = true$
21: **a**ctions:
22: IF $prelearned[l][p] \neq none$
23: THEN **store** $prelearned[l][p]$ **and** $v(p)$ for later notification
24: ELSE $prelearned[l][p] = v(p)$
25: $Learn(l) \triangleq$
26: **pre-conditions:**
27: $l \in learners$
28: $\exists Q$, Q is a quorum: $\forall a \in Q$ received (m, s), m = $\langle$"2b", $a, r, \_\rangle$ **and** $verify(a, m, s) = true$
29: **actions:**
30: LET $P \subset CF(r) : \forall p \in P$ received (m, s), m = $\langle$"2a", $p, r, \langle p, Nil\rangle\rangle$
31: LET $T \subset P : \forall p \in T$, $verify(p, m, s) = false$
32: LET $NQ \subset Q : \forall a \in NQ$ received $\langle$"2b", $a, r, noop\langle u, \langle q, W\rangle\rangle\rangle$
33: **if** $T \neq \emptyset$ **then** $\forall p \in T$, notify failure detector that p is byzantine
34: $Q2bVals = [v :$ received $\langle$"2b", $a, r, v\rangle$ from $a \in Q \setminus NQ\ ]$
35: $bn = [\langle u, Nil\rangle : u \in NQ]$
36: $w = \sqcap Q2bVals \bullet [\langle p, Nil\rangle : p \in T] \bullet bn$
37: $learned[l] = learned[l] \sqcup w$

Qualquer agente do protocolo é um agente bizantino em potencial, e portanto, precisa ser monitorado pelos demais e reportado ao detector de falhas se houver alguma suspeita, para que seja possível garantir o progresso e consistência do algoritmo. Uma forma de se assegurar que a denúncia de um agente bizantino não é uma ação bizantina, é provar ao

detector de falhas que o agente é bizantino apresentando a mensagem problemática como prova.

Abaixo descreveremos alguns possíveis cenários de agentes bizantinos tentando comprometer o funcionamento correto do protocolo e mostraremos como o uso de assinatura digital pode ajudar a corrigi-los.

5.3.1 Proposer Bizantino

A Figura 21 demonstra possíveis ações irregulares que um *collision-fast proposer* bizantino poderia tentar executar no protocolo. CFP_0 significa *collision-fast proposer* cujo número identificador é 0, no exemplo esse *collision-fast proposer* representa os votos de um *proposer* P_0. *A* denota *acceptors* e *L learners*, uma mensagem é ilustrada pela seta cheia informando o sentido em que a mensagem foi enviada e em seu "corpo" o identificador de quem enviou a mensagem e o conteúdo (valor), X por exemplo.

5.3.1.1 Votos Conflitantes

De acordo com protocolo especificado em (SCHMIDT; CAMARGOS; PEDONE, 2007), um *collision-fast proposer* de uma dada rodada que queira se abster de votar em uma instância pode propor *Nil* diretamente para um *learner* de mesma rodada, porém se assim o fizer, este *proposer* não pode propor outro valor nesta rodada.

Na Figura 21(a) o agente bizantino (*proposer*) propõe um valor X para um conjunto de *acceptors* e tenta propor *Nil* diretamente para algum *learner*. Tal ação pode causar conflito sobre o valor decidido, pois viola a propriedade de segurança do problema de *Consensus* Distribuído, que determina que um valor escolhido por algum processo, deve ser o mesmo escolhido por qualquer outro para uma mesma instância (LAMPORT, 2001).

Como o exemplo trata um caso onde algum *proposer* (P_0) é bizantino, este pode realizar ações não especificadas, ou não permitidas, pelo protocolo. Assim, se o *learner* receber o valor *Nil* de algum *collision-fast proposer* antes de algum outro valor diferente de *Nil* (e.g. X) do conjunto de *acceptors*, implicará que o valor mapeado para esse *collision-fast proposer* é nulo ($\{P_0 \rightarrow Nil\}$) para essa rodada dessa instância.

Porém, após receber um *v-mapping* $\{P_0 \rightarrow X\}$ do conjunto de *acceptors* para a mesma rodada e instância anterior, ocorreria um conflito, já que um valor já foi proposto, e mapeado para esse *proposer*. Logo, utilizando do método de assinatura digital, como especificado no Algoritmo 5.4 no método *Learn*, um *learner* L consegue verificar se o remetente é o mesmo que já lhe enviou um valor anterior (P_0 enviando *Nil*) e conclui que já recebeu uma mensagem desse remetente e que consequentemente há algo de errado. De posse de dois votos conflitantes e assinados, o learner pode provar ao detector de falhas que o agente é bizantino. Essa informação é passada ao *Collision-fast Oracle*, que

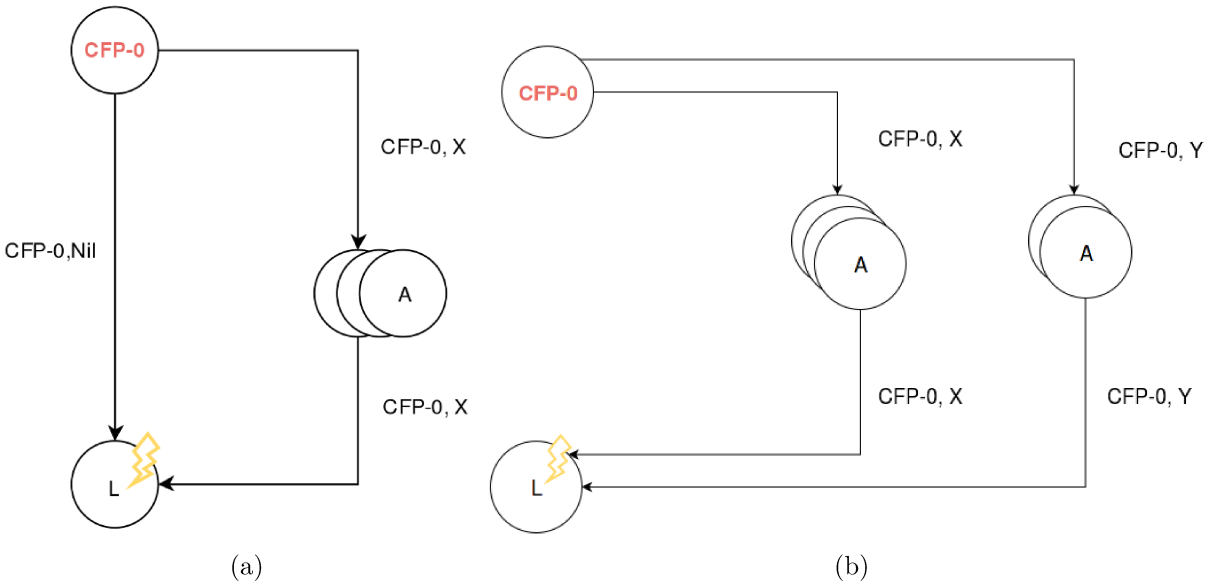


Figura 21 – Collision-fast proposer bizantino (CFP_0) propondo valores distintos para uma mesma rodada e instância.

excluirá o agente bizantino do conjunto de *collision-fast proposers* da próxima vez que for consultado.

Um *proposer* bizantino também poderia propor dois valores para conjuntos distintos de *acceptors*, na tentativa de formar *quorums* distintos, como mostrado na Figura 21(b). Porém, temos a exigência para formação de *quorum* citada na Seção 2.7 de que, dado quaisquer dois *quorums*, a intersecção entre eles não é vazia. Assim, se em algum momento o *learner* aprender algum dos dois valores (caso venha a formar *quorum*), garantidamente não o fará para o segundo valor. Novamente, tais votos, assinados, são provas do caráter malevolente do *proposer*.

Observe que, caso *proposers* bizantinos entrem em conluio e compartilhem suas chaves privadas, estes conseguirão apenas agir de forma equivalente a um mesmo *proposer* enviar mensagens distintas.

5.3.1.2 Personificação

A Figura 22 mostra um *collision-fast proposer* bizantino tentando se passar por outro, na tentativa de quebrar a decisão em apenas um valor, mas a tentativa é eventualmente percebida por um *acceptor*.

Isso é possível, porque o *acceptor*, por meio de uma unidade certificadora, confia nas informações sobre a real identidade dos participantes e consegue distinguir que recebeu informações diferentes de agentes que dizem ser CFP_1. Como apenas CFP_1 tem sua chave privada e a utiliza na assinatura das mensagens que envia, o *acceptor* consegue autenticá-las, o que não ocorre com as mensagens com a identidade falsificada enviadas por CFP_0, já que ele não conhece a chave privada do CFP_1, apenas a pública, não sendo possível

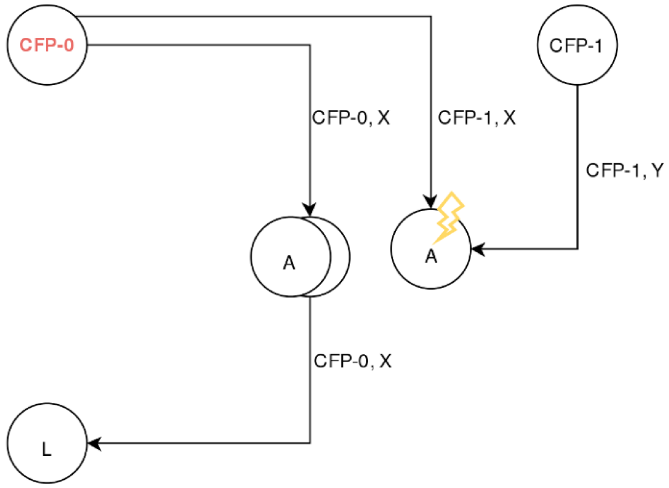


Figura 22 – CFP_0 bizantino tentando falsificar o voto de CFP_1.

obter uma via à outra. Nesse caso, pode não ser possível identificar o agente malicioso, mas isso não é um problema grave já que suas ações não afetam o resultado do protocolo, mesmo que possam afetar seu desempenho.

5.3.2 Coordenador Bizantino

Existe ainda o caso em que o coordenador pode ser um agente bizantino. Nesse cenário, o coordenador poderia, por exemplo, forçar constantes reconfigurações impedindo o algoritmo de atingir uma decisão. Ou ainda, iniciar uma nova rodada em que apenas ele éo *Collision-fast proposer* de todas as futuras instâncias, tendo apenas o seu valor, em algum momento, decidido, ou simplesmente ignorar qualquer valor informado pelos *acceptors* durante a *Phase*1, forçando novos aceites.

Nossa solução para o problema bizantino considera um coordenador confiável, deixando resolver *M-Consensus* onde um coordenador pode ser bizantino como um problema em aberto, que pretendemos explorar em trabalhos futuros. Essa restrição é feita pois um coordenador bizantino consegue controlar boa parte das funcionalidades do sistema, comprometendo o desempenho e a segurança da informação no serviço. Aqui propomos algumas ideias de como tal problema poderia ser resolvido:

- Durante a *Phase1* os *acceptors* podem ser obrigados a, além de responder com mensagens $Msg1B$ o coordenador da rodada, também enviarem tais mensagens para todos os demais *acceptors*, permitindo que estes também verifiquem as compatibilidades entre o que já foi aceito.

- Outra opção seria utilizar a Multi-Coordenação (CAMARGOS; SCHMIDT; PEDONE, 2007) como forma de se exigir um *quorum* de coordenadores para realizar alguma ação.

❑ Finalmente, o papel de coordenador pode ser rotacionado entre diversos proposers, usando critérios temporais bem como número de instâncias, limitando o poder do coordenador bizantino.

5.3.3 Acceptor Bizantino

Um *acceptor* bizantino poderia agir de várias formas para tentar danificar o sistemas. Por exemplo, ele poderia tentar modificar um voto. Contudo, dado o uso de assinaturas, esse ataque é trivialmente detectável. Assim, ele poderia simplesmente fingir que recebeu um voto (forjar aceites), mas como há a limitação de $f < n/3$ *acceptors* bizantinos, onde n é o número total de *acceptors*, estes aceites forjados nunca obteriam um *quorum*.

Contudo, estes votos forjados conflitariam com os votos reais, o que poderia levar a suspeitas de que os *proposers* enviaram votos conflitantes. Para solucionar esse caso, o aceite deve ser acompanhado de uma prova do voto, isto é, da mensagem enviada e assinada pelo *proposer*. Assim, em nossa solução, os votos, depois de aceitos, são repassados aos *learners*, impedindo que o *accceptor* aja nessa linha, como ilustrado na Figura 23.

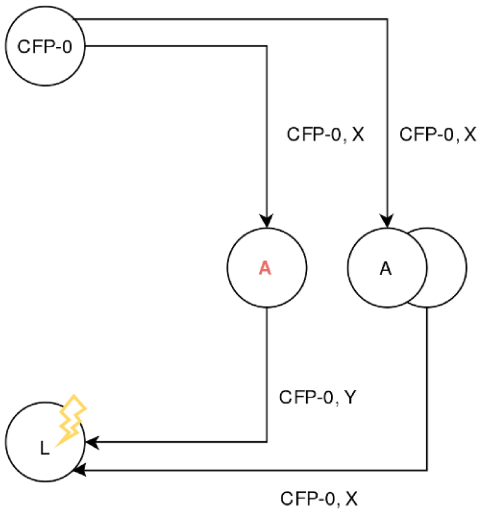


Figura 23 – Acceptor bizantino (em vermelho), impedido de modificar ou forjar votos.

5.3.4 Learner Bizantino

Se algum *learner* for bizantino, por a sua função no protocolo estar diretamente ligada com a consistência das respostas entregues na aplicação, tal consistência não poderia ser garantida, já que o *learner* pode escolher entregar para a aplicação o valor que assim desejar. Para evitar que isso ocorra, o learner poderia repassar os votos e aceites para a aplicação, que os verificaria. Contudo, devido à quebra da modularidade do protocolo, ao retrabalho imposto à aplicação, preferimos assumir uma improbabilidade de que o learner tenha sido comprometido sem que a aplicação não tenha e manter o protocolo simples.

5.3.5 Por quê quóruns maiores são necessário?

Considere um conluio entre um *proposer* CFP_b e *acceptor* A_b. Se maioria simples for utilizada, Figura 24 (a), então o *acceptor* bizantino poderia fazer parte de dois *quorums* (Q_1 e Q_2), em que aceita mensagens distintas (X e Y) do mesmo *proposer* bizantino, violando o acordo. Para evitar este cenário, um *quorum* tem que interceptar em pelo menos um processo correto de cada outro *quorum*. Isso exige que o *quorum* tenha tamanho $2f + 1$. Como já visto, $n > 3f$. Logo, cada *quorum* precisa ser maior que $\frac{2n}{3}$, como representado na Figura 24 (b).

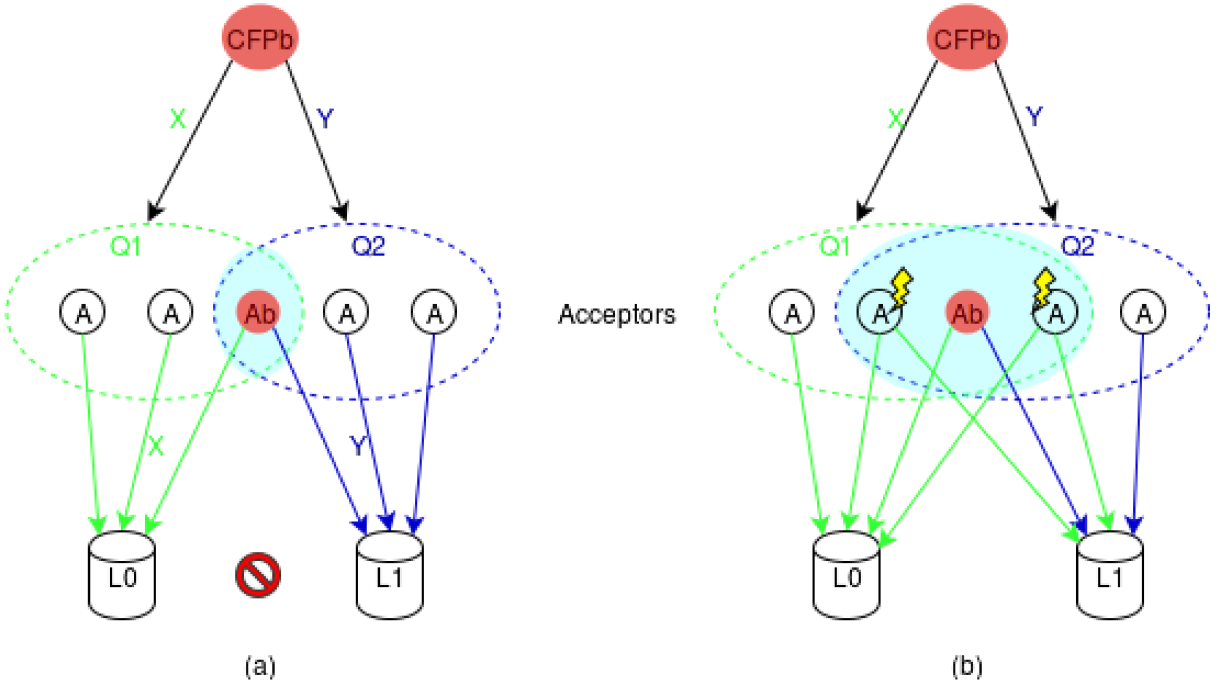


Figura 24 – Supondo um sistema com 5 *acceptors*. Temos que em (a) L_0 pode aprender X e L_1 aprender Y, vindos de diferentes *quorums* de tamanho 3. Já em (b), com um *quorum* de tamanho 4, L_0 consegue aprender X, ao passo que L_1 recebe provas de votos conflitantes de CFP_b, e pode provar que pelo menos esse *proposer* é bizantino e que propôs dois valores distintos.

5.4 Impossibilidade de Collision-fast Atomic Broadcast Bizantino Quiescente

Dizemos que um protocolo de difusão atômica é quiescente se existe um instante de tempo após o momento em que a última mensagem foi difundida, a partir do qual nenhuma mensagem trafega na rede. Esta propriedade denota que o algoritmo só causa custo de comunicação se estiver executando alguma tarefa útil.

Considere o cenário representado pela Figura 25, em que há dois *Collision-fast proposer* CFP_0 e CFP_1, e em que nenhuma falha, bizantina ou não, acontece. CFP_0 propõe X,

e de acordo com o protocolo, informa CFP_1 para que também vote. CFP_1 pode propor algum valor para A e os demais *CFPs* ou, se não tiver nenhum valor para propor nessa instância, *Nil* diretamente para L.

A possibilidade de algum deles ser bizantino implica que, se CFP_0 votou algum valor, e CFP_1 votou *Nil* direto para o *learner*, então o *learner* não pode aprender *Nil* imediatamente, pois CFP_1 pode ser bizantino. Logo, o *learner* precisa esperar pelo voto vindo via *acceptors*, que será referendado por uma maioria destes agentes.

Mas esse voto depende de algum cliente ter enviado algum valor para CFP_1, que propôs esse valor para ser votado. Como essa ação pode não ocorrer, o L pode ficar esperando por tempo indeterminado para finalizar a instância, completar o *v-mapping*.

Uma alternativa seria exigir que um *Collision-fast proposer* proponha um valor especial NO-OP caso receba uma mensagem *Msg2A* de outro *Collison-fast proposer* e não tenha um valor diferente de NO-OP ou *Nil* para propor. Esta abordagem, contudo, leva dois passos de comunicação para chegar (voto de NO-OP via *acceptor*), e como o voto de CFP_1 pode ter levado um passo de comunicação após o voto de CFP_0, então o *learner* só aprende o valor de CFP_1 três passos de comunicação após o início da instância.

Assim, conjecturamos que nenhum protocolo bizantino possa ser *collision fast* e quiescente, ou seja, decidir em dois passos de comunicação, na possível presença de agentes bizantinos, e de forma quiescente.

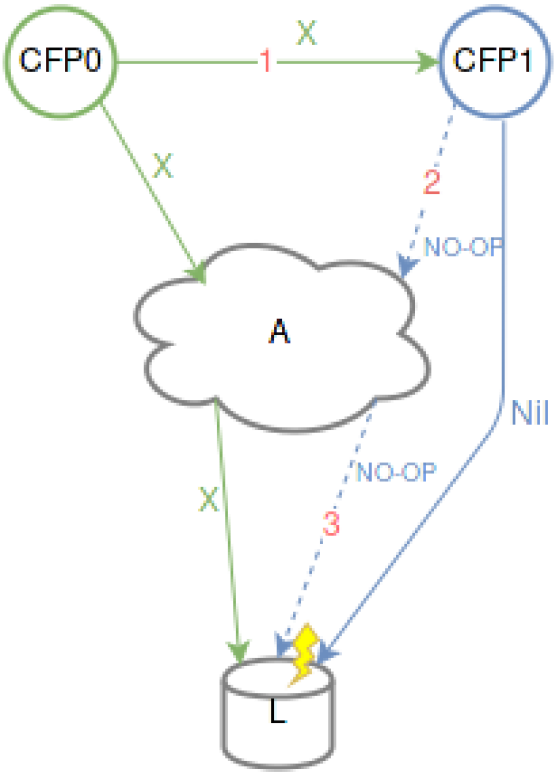


Figura 25 – Quantidade de passos de comunicação necessários para se decidir um *v-mapping* completo. Cada seta indica um passo de comunicação.

O protocolo apresentado nos Algoritmos 5.4, 5.5, e 5.6 pode ser enganado por um *collision-fast proposer* bizantino agindo como na Figura 25, e corrigir tal deficiência é um trabalho em progresso. A principal abordagem sendo considerada para fazê-lo é entregar o *Nil* otimisticamente para a aplicação e forçar o *proposer* a enviar um comando NO-OP (como por exemplo: $\{P_1 \rightarrow \{P_0 \rightarrow X\}\}$) contendo uma *prova* que justifique o voto *Nil* feito. Essa prova confirmaria o voto *Nil*, que seria então conservadoramente entregue à aplicação, e provaria que o *proposer* não votou com o intuito de causar perturbação no sistema para tentar impedir avanço.

Capítulo 6

Conclusões e Trabalhos Futuros

Neste trabalho apresentamos a *framework* que desenvolvemos para avaliação de protocolos de acordo, descrevendo as decisões de design tomadas e os desafios encontrados durante o processo de se codificar uma especificação, ainda mais se tratando de protocolos notoriamente difíceis de se implementar da forma correta.

Mostramos que para se implementar o protocolo *Collision-fast Atomic Broadcast*, proposto em (SCHMIDT; CAMARGOS; PEDONE, 2007), é necessário uma série de mecanismos adicionais para tratar os cenários práticos de um sistema de máquinas de estado replicadas. E até onde sabemos, essa é a única implementação existente do protocolo *Collision-fast Atomic Broadcast*[1], disponibilizada como *open source*, e que de acordo com nossa avaliação, apresentou bom desempenho em comparação com os demais.

Além disso, alteramos o protocolo para possibilitar execução especulativa dos comandos. Detalhamos o algoritmo que permite que o *Collision-fast Atomic Broadcast* entregue mensagens de forma especulativa e o implementamos em nosso *framework*. Esse método possibilita a entrega de uma possível decisão mais rapidamente, ao passo que pode comprometer a resiliência do sistema.

Os experimentos conduzidos no Emulab mostraram que o *Collison-fast Paxos* consegue manter menor latência e maior vazão que o *Paxos*, e em grade parte dos casos que o *Multi Ring Paxos* nos cenários avaliados. Foram avaliados o *Paxos*, o *Collision-fast Paxos* e o *Ring Paxos*, utilizando um sistema de replicação de máquinas de estado especialmente desenvolvido para os testes. Os dois primeiros foram implementados na nossa *framework*, o último está disponível na *Libmcad*, biblioteca em Java utilizada nos testes.

Não obstante, no Capítulo 5 propomos uma extensão do protocolo *Collision-fast Atomic Broadcast* que tolera falhas bizantinas, e conjecturamos ser impossível criar um protocolo bizantino que seja *Collision-fast*. Apresentamos os detalhes do algoritmo proposto, descrevendo como o uso de assinatura digital auxilia na detecção da falhas bizantinas no nosso algoritmo.

[1] <https://github.com/r0qs/cfabcast>

Atualmente estamos trabalhando para melhorar a arquitetura da *framework* desenvolvida, que já está sendo utilizada em outro trabalho de pesquisa em problemas de acordo. Também pretendemos implementar e avaliar o *Collision-fast Atomic Broadcast* em cenários de falhas que exijam reconfiguração, bem como implementar o modelo bizantino e estender nosso ambiente de avaliação de desempenho. Esperamos que esse trabalho traga mais visibilidade para esse protocolo que possui mesmas garantias que o *Paxos* porém apresenta, teórica, e agora experimentalmente, melhor desempenho que o *Paxos*. Sendo uma alternativa de protocolo *fast* que não sofre do problema de colisões.

6.1 Trabalhos Futuros

Na implementação do nossa *framework* foi utilizado um conjunto de ferramentas e bibliotecas desenvolvido especialmente para construção de aplicações orientadas a mensagens, altamente concorrentes, distribuídas e resilientes na JVM, chamada Akka, que possui uma gama de parâmetros de configuração que afetam o desempenho e necessitam ser exploradas mais a fundo.

Além disso, o uso da nossa *framework* em outro trabalho de pesquisa trouxe a tona vários *bugs* de implementação, muitos dos quais já foram corrigidos, porém ainda há alguns que requerem mudanças mais bruscas, como a substituição da nossa abstração de intervalos de instâncias e melhoria na estrutura de dados que utilizamos para contabilizar os *quorums*. Estes são os próximos problemas que pretendemos atacar, para manter uma base de implementação sólida, munida de testes unitários e de integração, que permita a extensão de protocolos de forma fácil e prática.

Assim, conseguiremos construir um ambiente ideal para os próximos passos, realizar testes de desempenho considerando falhas no sistema. Logo, precisaremos desenvolver mecanismos de simulação de falhas e sistematizar cenários de reconfigurações a serem testados.

Tal ambiente, possibilita implementar vários protocolos de acordo e testa-los sob mesmas condições, apurando de forma mais precisa o impacto prático de cada protocolo, avaliados em ambientes reais. Assim, além de implementar mais protocolos de acordo, também pretendemos melhorar as métricas coletadas, utilizando de mais informações sobre o sistema nas analises futuras.

Por fim, nosso intuito também é estudar melhor problemas de acordo sob a ótica de falhas bizantinas. Para tanto, pretendemos implementar na nossa *framework* nossa versão do algoritmo, que tolera falhas bizantinas, mas que não é *collision-fast*, já que conjecturamos não ser possível existir protocolo assim.

6.2 Contribuições em Produção Bibliográfica

Em 2015 tivemos um artigo do tipo *fast-abstract* aceito na *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) 2015*, intitulado *Implementation and evaluation of Collision-fast Atomic Broadcast protocols*, no qual descrevemos parte da arquitetura da *framework* (SARAMAGO; CAMARGOS, 2015).

Referências

ABRAHAM, I.; AMIT, Y.; DOLEV, D. Optimal resilience asynchronous approximate agreement. In: HIGASHINO, T. (Ed.). **Principles of Distributed Systems**. Springer Berlin Heidelberg, 2005, (Lecture Notes in Computer Science, v. 3544). p. 229–239. ISBN 978-3-540-27324-0. Disponível em: <http://dx.doi.org/10.1007/11516798_17>.

BRUCE, K. B.; ODERSKY, M.; WADLER, P. A statically safe alternative to virtual types. In: . [S.l.]: Springer-Verlag, 1997. p. 523–549.

BURROWS, M. **The Chubby lock service for loosely-coupled distributed systems**. 2006. Disponível em: <http://research.google.com/archive/chubby.html>.

BUTCHER, P. **Seven Concurrency Models in Seven Weeks: When Threads Unravel**. 1st. ed. [S.l.]: Pragmatic Bookshelf, 2014. ISBN 1937785653, 9781937785659.

CAMARGOS, L. J.; SCHMIDT, R. M.; PEDONE, F. Multicoordinated paxos. In: **Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing**. New York, NY, USA: ACM, 2007. (PODC '07), p. 316–317. ISBN 978-1-59593-616-5. Disponível em: <http://doi.acm.org/10.1145/1281100.1281150>.

CASTRO, M.; LISKOV, B. Practical byzantine fault tolerance. In: **Proceedings of the Third Symposium on Operating Systems Design and Implementation**. Berkeley, CA, USA: USENIX Association, 1999. (OSDI '99), p. 173–186. ISBN 1-880446-39-1. Disponível em: <http://dl.acm.org/citation.cfm?id=296806.296824>.

CHANDRA, T. D.; GRIESEMER, R.; REDSTONE, J. Paxos made live: An engineering perspective. In: **Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing**. New York, NY, USA: ACM, 2007. (PODC '07), p. 398–407. ISBN 978-1-59593-616-5. Disponível em: <http://doi.acm.org/10.1145/1281100.1281103>.

CHANDRA, T. D.; HADZILACOS, V.; TOUEG, S. The weakest failure detector for solving consensus. **J. ACM**, ACM, New York, NY, USA, v. 43, n. 4, p. 685–722, jul. 1996. ISSN 0004-5411. Disponível em: <http://doi.acm.org/10.1145/234533.234549>.

CHANDRA, T. D.; TOUEG, S. Unreliable failure detectors for reliable distributed systems. **Journal of the ACM**, v. 43, p. 225–267, 1995.

CRISTIAN, F. Reaching agreement on processor group membership in synchronous distributed systems. **Distributed Computing**, v. 4, p. 175–187, 1991.

DECANDIA, G. et al. Dynamo: amazon's highly available key-value store. In: **IN PROC. SOSP**. [S.l.: s.n.], 2007. p. 205–220.

DÉFAGO, X.; SCHIPER, A.; URBÁN, P. Total order broadcast and multicast algorithms: Taxonomy and survey. **ACM Computing Surveys**, v. 36, p. 2004, 2003.

DÉFAGO, X. et al. The phi accrual failure detector. In: **RR IS-RR-2004-010, Japan Advanced Institute of Science and Technology**. [S.l.: s.n.], 2004. p. 66–78.

DU, J. et al. Clock-RSM: Low-latency inter-datacenter state machine replication using loosely synchronized physical clocks. In: **44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014**. [s.n.], 2014. p. 343–354. Disponível em: <http://dx.doi.org/10.1109/DSN.2014.42>.

DWORK, C.; LYNCH, N.; STOCKMEYER, L. Consensus in the presence of partial synchrony. **J. ACM**, ACM, New York, NY, USA, v. 35, n. 2, p. 288–323, abr. 1988. ISSN 0004-5411. Disponível em: <http://doi.acm.org/10.1145/42282.42283>.

EIDE, E.; STOLLER, L.; LEPREAU, J. An experimentation workbench for replayable networking research. In: **In Proceedings of the Symposium on Networked System Design and Implementation**. [S.l.: s.n.], 2007.

EUGSTER, P. T. et al. Epidemic information dissemination in distributed systems. **Computer**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 37, n. 5, p. 60–67, maio 2004. ISSN 0018-9162. Disponível em: <http://dx.doi.org/10.1109/MC.2004.1297243>.

FISCHER, M. J.; LYNCH, N. A.; PATERSON, M. S. **Impossibility of Distributed Consensus with One Faulty Process**. 1985.

GILBERT, S.; LYNCH, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. **SIGACT News**, ACM, New York, NY, USA, v. 33, n. 2, p. 51–59, jun. 2002. ISSN 0163-5700. Disponível em: <http://doi.acm.org/10.1145/564585.564601>.

HEWITT, C.; BISHOP, P.; STEIGER, R. A universal modular actor formalism for artificial intelligence. In: **Proceedings of the 3rd International Joint Conference on Artificial Intelligence**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1973. (IJCAI'73), p. 235–245. Disponível em: <http://dl.acm.org/citation.cfm?id=1624775.1624804>.

KIRSCH, J.; AMIR, Y. Paxos for system builders: An overview. In: **Proceedings of the 2Nd Workshop on Large-Scale Distributed Systems and Middleware**. New York, NY, USA: ACM, 2008. (LADIS '08), p. 3:1–3:6. ISBN 978-1-60558-296-2. Disponível em: <http://doi.acm.org/10.1145/1529974.1529979>.

LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. **Commun. ACM**, ACM, New York, NY, USA, v. 21, n. 7, p. 558–565, jul. 1978. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/359545.359563>.

_____. The part-time parliament. **ACM TRANSACTIONS ON COMPUTER SYSTEMS**, v. 16, n. 2, p. 133–169, 1998.

_____. Paxos Made Simple. **SIGACT News**, ACM, New York, NY, USA, v. 32, n. 4, p. 51–58, dez. 2001. ISSN 0163-5700. Disponível em: <http://research.microsoft.com/users/lamport/pubs/paxos-simple.pdf>.

_____. Fast paxos. **Distributed Computing**, Springer-Verlag, v. 19, n. 2, p. 79–103, 2006. ISSN 0178-2770. Disponível em: <http://dx.doi.org/10.1007/s00446-006-0005-x>.

_____. Lower bounds for asynchronous consensus. **Distributed Computing**, Springer-Verlag, v. 19, n. 2, p. 104–125, 2006. ISSN 0178-2770. Disponível em: <http://dx.doi.org/10.1007/s00446-006-0155-x>.

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. Disponível em: <http://doi.acm.org/10.1145/357172.357176>.

LAMPSON, B. How to build a highly available system using consensus. In: BABAOGLU, O.; MARZULLO, K. (Ed.). **Distributed Algorithms**. Springer Berlin Heidelberg, 1996, (Lecture Notes in Computer Science, v. 1151). p. 1–17. ISBN 978-3-540-61769-3. Disponível em: <http://dx.doi.org/10.1007/3-540-61769-8_1>.

MAO, Y.; JUNQUEIRA, F. P.; MARZULLO, K. Mencius: Building efficient replicated state machines for wans. In: **Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation**. Berkeley, CA, USA: USENIX Association, 2008. (OSDI'08), p. 369–384. Disponível em: <http://dl.acm.org/citation.cfm?id=1855741.1855767>.

MARANDI, P. J.; PRIMI, M.; PEDONE, F. Multi-ring paxos. In: **Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)**. Washington, DC, USA: IEEE Computer Society, 2012. (DSN '12), p. 1–12. ISBN 978-1-4673-1624-8. Disponível em: <http://dl.acm.org/citation.cfm?id=2354410.2355144>.

MARANDI, P. J. et al. Ring Paxos: A High-Throughput Atomic Broadcast Protocol. 2010. Disponível em: <http://www.inf.usi.ch/phd/schiper/research/DSN10.pdf>.

MIZERANY, B.; RARICK, K. **Doozer**. 2011. Disponível em: <https://github.com/ha/doozerd>.

PROKOPEC, A. et al. Concurrent tries with efficient non-blocking snapshots. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 47, n. 8, p. 151–160, fev. 2012. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/2370036.2145836>.

RAMASAMY, H. V.; CACHIN, C. Parsimonious asynchronous byzantine-fault-tolerant atomic broadcast. In: **Proceedings of the 9th International Conference on Principles of Distributed Systems**. Berlin, Heidelberg: Springer-Verlag, 2006. (OPODIS'05), p. 88–102. ISBN 3-540-36321-1, 978-3-540-36321-7. Disponível em: <http://dx.doi.org/10.1007/11795490_9>.

RENESSE, R. van; MINSKY, Y.; HAYDEN, M. A gossip-style failure detection service. In: **Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing**. London, UK, UK: Springer-Verlag, 1998. (Middleware '98), p. 55–70. ISBN 1-85233-088-0. Disponível em: <http://dl.acm.org/citation.cfm?id=1659232.1659238>.

SARAMAGO, R. Q.; CAMARGOS, L. J. Implementation and evaluation of collision-fast atomic broadcast protocols (fast abstract). In: **45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2015, Rio de Janeiro, Brazil, June 22-25, 2015**. [S.l.: s.n.], 2015.

SCHMIDT, R.; CAMARGOS, L.; PEDONE, F. **On Collision-fast Atomic Broadcast**. [S.l.], 2007. Disponível em: <http://infoscience.epfl.ch/getfile.py?recid=100857>.

_____. Collision-fast atomic broadcast. In: **Proceedings of the 2014 IEEE 28th International Conference on Advanced Information Networking and Applications**. Washington, DC, USA: IEEE Computer Society, 2014. (AINA '14), p. 1065–1072. ISBN 978-1-4799-3630-4.

SUBRAMANIYAN, R. et al. Gems: Gossip-enabled monitoring service for scalable heterogeneous distributed systems. **Cluster Comput**, 2006.

WARD, J. S.; BARKER, A. Self managing monitoring for highly elastic large scale cloud deployments. In: **Proceedings of the Sixth International Workshop on Data Intensive Distributed Computing**. New York, NY, USA: ACM, 2014. (DIDC '14), p. 3–10. ISBN 978-1-4503-2913-2. Disponível em: <http://doi.acm.org/10.1145/2608020.2608022>.

ZHAO, W. Fast paxos made easy: Theory and implementation. **Int. J. Distrib. Syst. Technol.**, IGI Global, Hershey, PA, USA, v. 6, n. 1, p. 15–33, jan. 2015. ISSN 1947-3532. Disponível em: <http://dx.doi.org/10.4018/ijdst.2015010102>.