Enrique Sampaio dos Santos

Validação Distribuída do Algoritmo Paxos no Modelo Arbitrário Não Malicioso

Sorocaba, SP

3 de Novembro de 2020

Enrique Sampaio dos Santos

Validação Distribuída do Algoritmo Paxos no Modelo Arbitrário Não Malicioso

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação (PPGCC-So) da Universidade Federal de São Carlos como parte dos requisitos exigidos para a obtenção do título de Mestre em Ciência da Computação. Linha de pesquisa: Engenharia de Software e Sistemas de Computação.

Universidade Federal de São Carlos – UFSCar

Centro de Ciências em Gestão e Tecnologia – CCGT

Programa de Pós-Graduação em Ciência da Computação – PPGCC-So

Orientador: Prof. Dr. Gustavo Maciel Dias Vieira

Sorocaba, SP

3 de Novembro de 2020

Sampaio dos Santos, Enrique

Validação Distribuída do Algoritmo Paxos no Modelo Arbitrário Não Malicioso / Enrique Sampaio dos Santos -- 2020.
67f.

Dissertação (Mestrado) - Universidade Federal de São Carlos, campus Sorocaba, Sorocaba
Orientador (a): Prof. Dr. Gustavo Maciel Dias Vieira
Banca Examinadora: Prof.ª Dr.ª Islene Calciolari, Prof.ª Dr.ª Yeda Regina Venturini
Bibliografia

1. Algoritmos Distribuídos. 2. Tolerância a Falhas. 3. Paxos. I. Sampaio dos Santos, Enrique. II. Título.

Ficha catalográfica desenvolvida pela Secretaria Geral de Informática (SIn)

DADOS FORNECIDOS PELO AUTOR

Bibliotecário responsável: Maria Aparecida de Lourdes Mariano - CRB/8 6979

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências em Gestão e Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Defesa de Dissertação de Mestrado do candidato Enrique Sampaio dos Santos, realizada em 03/11/2020.

Comissão Julgadora:

Prof. Dr. Gustavo Maciel Dias Vieira (UFSCar)

Profa. Dra. Islene Calciolari Garcia (UNICAMP)

Profa. Dra. Yeda Regina Venturini (UFSCar)

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

O Relatório de Defesa assinado pelos membros da Comissão Julgadora encontra-se arquivado junto ao Programa de Pós-Graduação em Ciência da Computação.

Dedico este trabalho aos meus pais Antonio Enrique dos Santos e Maria Izabel Sampaio dos Santos que sempre me deram e ainda dão a base e as condições necessárias para me dedicar e priorizar os estudos;

Às minhas irmãs Izabela Sampaio dos Santos e Ana Paula Sampaio dos Santos e ao meu irmão Eduardo Sampaio dos Santos pelo incentivo e exemplo;

À minha namorada Jeniffer Bagatin por todo incentivo, suporte, desabafos e conselhos sem os quais nada disso seria possível.

Aos meus colegas de trabalho Alan Castro Silva, Eduardo Rodrigues da Cruz e Marcelo Kenji Murosaki Marczuk por todo o apoio e por tornar mais fácil a conciliação entre os trabalhos na empresa e na pesquisa;

A todos os amigos que fiz nos grupos de pesquisa LERIS e LaSID que me incentivaram a ingressar no mestrado e tornaram a jornada mais fácil;

A todos que direta ou indiretamente fizeram parte dessa jornada.

Agradecimentos

Agradeço,

ao meu orientador Prof. Dr. Gustavo Maciel Dias Vieira pela colaboração, opiniões e revisão durante o desenvolvimento de todo o trabalho e escrita de projeto, artigo e dissertação.

ao colega de pesquisa Rodrigo Rocco Barbieri pela colaboração em artigo e produção de um trabalho que serviu como alicerce para esta dissertação.

à banca avaliadora composta pelas professoras Dr.ª Islene Calciolari Garcia e Dr.ª Yeda Regina Venturini por terem aceitado participar e contribuírem com suas opiniões durante a qualificação do projeto e agora em sua defesa.

"As pessoas pensam na educação como algo que podem terminar (...). Você tem todo mundo ansioso para não aprender mais, e você os envergonha depois de voltar a aprender (...).

Se você gosta de aprender, não há razão para parar em uma determinada idade. As pessoas não param de fazer coisas que gostam de fazer só porque atingem uma certa idade. O que é empolgante é o processo de se ampliar, de saber que agora há uma pequena faceta extra do universo que você conhece e pode pensar e entender.

(Isaac Asimov)

Resumo

Algoritmos distribuídos têm sido cada vez mais utilizados por fatores comerciais e técnicos, principalmente quando o intuito é obter escalabilidade ou alta disponibilidade dos dados. Neste último ponto, destacam-se os algoritmos em que um conjunto de processos interagem através de trocas de mensagens a fim de executarem as mesmas ações e se manter no mesmo estado, chamados de algoritmos de replicação. Entretanto, para atender essa alta demanda não basta tais algoritmos serem funcionais como também devem ser tolerantes a falhas, a fim de evitar corrupções dos dados. Tolerância a falhas em algoritmos distribuídos não é um tópico trivial, e algoritmos que toleram falhas arbitrárias tendem a ser custosos e de implementação complexa. A fim de tolerar diversas categorias de falhas mantendo a complexidade baixa, foi criado o modelo de falhas arbitrárias não maliciosas, onde apenas falhas causadas por invasões intencionais ao ambiente não são toleradas. Esta dissertação propõe um mecanismo de validação distribuída para o algoritmo Paxos a fim de garantir as propriedades deste modelo, tornando-o tolerante a falhas arbitrárias não maliciosas.

Palavras-chaves: Algoritmos distribuídos. Tolerância a falhas. Falhas arbitrárias. Não malicioso. Falhas benignas. Validação distribuída.

Abstract

Distributed algorithms have been increasingly used because of commercial and technical factors, especially when the aim is to obtain scalability or high availability of data. Regarding high scalability, stands out the algorithms in which a set of processes interact through the exchange of messages in order to perform the same actions and remain in the same state, called replication algorithms. However, to meet this high demand, it is not enough for such algorithms to be functional, but they must also be fault tolerant, in order to avoid data corruption. Fault tolerance in distributed algorithms is not a trivial topic, and algorithms that tolerate arbitrary failure tend to be costly and complex to implement. In order to tolerate several fault types while keeping the complexity low, the model of arbitrary non-malicious faults was created, where only faults caused by intentional invasions to the environment are not tolerated. This dissertation proposes a distributed validation mechanism for the Paxos algorithm in order to guarantee the properties of the model, making it tolerant to non-malicious arbitrary faults.

Key-words: Distributed algorithms. Fault tolerance. Arbitrary faults. No-malicious. Benign faults. Distributed validation.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

PBFT	Practical Byzantine Fault Tolerance
RAM	Random Access Memory
CPU	Central Processing Unit
JDK	Java Development Kit
JRE	Java Runtime Environment
Op/s	Operações por segundo

Sumário

Introdução

Com o advento da computação nas mais diversas áreas do conhecimento e a produção cada vez maior de dados para processamento, a necessidade de poder computacional aumentou consideravelmente nos últimos anos. Uma forma de adquirir tal poder com uma escalabilidade horizontal é através de *sistemas distribuídos*.

De forma simplificada, um sistema distribuído consiste em uma série de máquinas compartilhando seus recursos através de uma rede, de forma que um processamento complexo possa ter suas tarefas divididas entre as diversas máquinas que interagem entre si para chegar a um resultado comum. Esse tipo de sistema é frequentemente utilizado quando desejamos aumentar o número de processos concorrentes em aplicações paralelizáveis sem precisar investir em super processadores com muitos núcleos, que costumam ser caros, podendo utilizar *hardwares* mais modestos e baratos em maior número. Além disso, por trazer consigo o benefício da replicação dos dados e do seu processamento, essa abordagem também vem sendo amplamente utilizada em sistemas que necessitam de redundância. Nesse cenário, destacam-se os *algoritmos de replicação*, onde o processamento dos dados é feito de forma idêntica em diversos processos a fim de chegar em um resultado consistente e com alta disponibilidade.

Algoritmos de replicação funcionam como diversas máquinas de estados deterministas realizando as mesmas ações, logo, sempre obtendo os mesmos estados. Como essas máquinas executam em processos diferentes, o isolamento entre elas permite uma tolerância a falhas maior do que de um processo independente, já que mesmo que algumas réplicas falhem os estados estarão disponíveis através das remanescentes (SCHNEIDER, 1990). No entanto, manter todas essas máquinas de estados idênticas não é uma tarefa trivial, ainda mais se considerado o cenário em que os novos estados podem surgir de qualquer uma das máquinas, devendo ser propagados para todas as outras e na ordem correta. A fim de resolver este problema e promover uma sequência de estados única entre todas as réplicas pode-se usar os *algoritmos de consenso* (LAMPORT, 1998).

Dentre os algoritmos de consenso, podemos citar como um dos mais difundidos o algoritmo *Paxos* (LAMPORT, 1998), onde os processos do sistema passam por diversas rodadas e trocas de mensagens para chegar a um acordo, que nesse cenário significa uma sequência única de estados para as máquinas de estados replicadas.

Sistemas distribuídos, assim como qualquer ambiente computacional, podem possuir *falhas*. Podemos considerar falhas como propriedades internas ou externas do ambiente que, quando presentes, inviabilizam a confiabilidade no serviço, pois o mesmo pode deixar de funcionar corretamente em determinadas situações, conhecidas ou não. Uma falha pode

nunca se manifestar, e nesse caso é classificada como uma falha *dormente*, ou então gerar um *erro*, e então é chamada de falha *ativa*. Erros são desvios no estado correto do serviço, que passa a funcionar de forma inesperada, e torna-se um *serviço defeituoso* (AVIZIENIS et al., 2004).

As falhas podem ser divididas em diversas categorias de acordo com seu comportamento. Entre as principais, existe a categoria em que o processo defeituoso torna-se inoperante, e também aquela onde o processo para de enviar mensagens aos demais processos do ambiente. Por fim, há ainda falhas em que o processo tem seu funcionamento interrompido mas se recupera após um período de tempo. Chamamos de falhas arbitrárias o conjunto em que todas as categorias de falhas estão contidas (CACHIN; GUERRAOUI; RODRIGUES, 2011).

Computadores tolerantes a falhas costumam ser mais caros e lentos do que máquinas comuns, o que justifica o uso dessa última categoria em *data centers* e outros ambientes distribuídos. No entanto, máquinas comuns nesses ambientes estão sujeitas a falhas arbitrárias como *bit-flips* e *stuck-at bits*, falhas que tendem a aumentar com as novas gerações de *hardware*. Uma forma de solucionar tal problema é através da implementação de algoritmos capazes de tornar um sistema distribuído tolerante a falhas (BEHRENS; WEIGERT; FETZER, 2013; CACHIN; GUERRAOUI; RODRIGUES, 2011; CORREIA et al., 2012; SCHNEIDER, 1990). Tais algoritmos já existem, e em sua forma mais completa são conhecidos como algoritmos tolerantes a falhas Bizantinas (CASTRO; LISKOV et al., 1999).

Algoritmos tolerantes a falhas Bizantinas, como o PBFT (*Practical Byzantine Fault Tolerance*) (CASTRO; LISKOV et al., 1999), são capazes de tolerar falhas em ambientes distribuídos. Entretanto, a alta complexidade de entendimento e de avaliar a correção desses algoritmos, junto à necessidade de *hardwares* poderosos para suportá-los que implicam em alto custo, tornam seu uso pouco difundido (BEHRENS; WEIGERT; FETZER, 2013).

Atualmente diversas ferramentas voltadas ao cenário de sistemas distribuídos já oferecem tolerância a falhas quando algum processo simplesmente para de funcionar sem acarretar em envio de mensagens incorretas, entretanto, nem sempre esse é o caso (CORREIA et al., 2012). Falhas não toleradas podem emitir mensagens compreensíveis, porém com conteúdo incorreto, ocasionando o funcionamento incorreto do sistema como um todo. Falhas como essas já causaram problemas em ambientes consolidados, como na *Amazon S3*, onde um único *bit* corrompido em um conjunto de mensagens passou despercebido e fez com que todo o ambiente precisasse ser reiniciado para voltar a funcionar corretamente (BEHRENS; WEIGERT; FETZER, 2013; CORREIA et al., 2012).

O principal motivo que explica o alto custo computacional dos algoritmos tolerantes a falhas Bizantinas são as redundâncias necessárias para realizar as validações de segurança.

Com o intuito de prover uma solução com menor complexidade de implementação e que necessite de menos recursos físicos do que as soluções tradicionais, uma abordagem foi a diminuição do escopo para o modelo computacional de falhas arbitrárias não maliciosas, ou seja, todas falhas que não são intencionalmente causadas por terceiros (BEHRENS; WEIGERT; FETZER, 2013).

Foi utilizado como base para o algoritmo desenvolvido durante esta pesquisa o trabalho apresentado por Barbieri (2016), que descreve uma implementação capaz de tolerar falhas arbitrárias não maliciosas em ambientes distribuídos. Nela, quando uma falha é detectada, o processo defeituoso é retirado do ambiente sem que envie mensagens errôneas aos demais, evitando assim que um erro possa se propagar, e o sistema continua funcionando normalmente. O problema dessa abordagem é que a detecção de falhas que ocorrem nos componentes do algoritmo Paxos acontece de tal maneira que, até que a mesma seja devidamente tolerada, o processo divergente autor da falha já teve a oportunidade de enviar mensagens às demais réplicas e participar de rodadas de decisão. Como será apresentado nas seções seguintes, tal comportamento não está em conformidade com as propriedades do modelo de falha não maliciosa, já que o envio de mensagens por parte de um processo defeituoso pode implicar em estados corrompidos serem propagados.

Levando em consideração tal problema, esta dissertação de mestrado propõe uma nova validação no algoritmo Paxos, substituindo a apresentada por Barbieri (2016), que se propõe a evitar que processos defeituosos consolidem qualquer estado corrompido e continuem participando de rodadas de consenso, tornando o algoritmo em conformidade com o modelo de falhas arbitrárias não maliciosas. O restante da dissertação está organizada da seguinte forma: o Capítulo 1 apresenta a contextualização bibliográfica. As implementações realizadas são detalhadas no Capítulo 2 e os experimentos executados para validar a proposta, assim como os resultados obtidos, são apresentados no Capítulo 3. Por fim, uma conclusão sobre todo o processo é apresentada assim como uma breve análise sobre possíveis trabalhos futuros.

1 Revisão Bibliográfica

Para o entendimento da dissertação é necessário compreender alguns conceitos fundamentais da área de sistemas distribuídos, além de entender o funcionamento de alguns algoritmos e ferramentas que serão utilizadas como base para a proposta.

1.1 Modelos Computacionais

O primeiro conceito que define as características de um sistema e que nos ajuda a entender seu comportamento é o de modelo computacional. Um modelo computacional consiste, basicamente, de uma série de abstrações que definem o funcionamento do sistema, seu comportamento em caso de falha e suas limitações.

1.1.1 Modelos clássicos

Existem modelos computacionais amplamente difundidos e utilizados no âmbito de sistemas distribuídos, entre eles os modelos de falha e recuperação e de falhas arbitrárias (CACHIN; GUERRAOUI; RODRIGUES, 2011).

1.1.1.1 Modelo de falha e recuperação

No modelo de falha e recuperação supõe-se que processos podem parar devido a uma falha e se recuperar mais tarde, voltando a participar do sistema. Durante a interrupção da operação o processo não participa da computação, e ele não se desvia da especificação do algoritmo em nenhum momento que esteja operando. Os algoritmos criados para esse modelo devem considerar que um processo, após se recuperar, possa ter perdido o registro de tudo que ocorreu antes de parar, podendo utilizar memória persistente como uma solução para esse problema. (CACHIN; GUERRAOUI; RODRIGUES, 2011).

1.1.1.2 Modelo de falhas arbitrárias

Quando não há como prever o comportamento de um processo ao apresentar uma falha, ou o tipo de falha que pode ocorrer, o modelo de falhas arbitrárias é a opção mais coerente. Neste modelo não é feita nenhuma suposição sobre as falhas, suas origens e consequências. Por exemplo, um processo pode parar de funcionar intermitentemente, enviar mensagens incorretas, omitir mensagens, etc (CACHIN; GUERRAOUI; RODRIGUES, 2011).

1.1.2 Modelo arbitrário não malicioso

Para o trabalho desenvolvido foi utilizado um modelo derivado dos modelos clássicos chamado de *falhas arbitrárias não maliciosas* (BEHRENS; WEIGERT; FETZER, 2013), em que todo tipo de falha deve ser tolerado, exceto aquelas criadas de forma maliciosa, ou seja, de forma proposital por meio de ataques externos ou internos. Todos modelos citados supõem que o sistema é formado por um conjunto de N processos.

Dentre todos os tipos de falha, as mais difíceis de se tolerar são as chamadas falhas maliciosas, já que as mesmas são causadas de forma intencional e podem ter as mais diversas características e comportamentos, podendo inclusive serem desenvolvidas de modo que o processo simule um comportamento correto. Desconsiderando tais falhas, restam aquelas que não são "planejadas", ou sejam, ocorrem de forma espontânea. Essas falhas espontâneas obedecem uma distribuição estatística que é especificada pelas propriedades *sem personificação* e *honra* (BEHRENS; WEIGERT; FETZER, 2013).

1.1.2.1 Sem personificação: O ambiente nunca cria mensagens válidas (exceto duplicatas)

A propriedade "sem personificação" define que o ambiente nunca cria espontaneamente mensagens válidas além de duplicatas. Um invasor poderia analisar o conjunto de mensagens válidas do sistema e gerar falhas através de mensagens arbitrárias fraudulentas porém válidas. Sem a variável do ataque humano, a probabilidade desse tipo de mensagem ser criada do nada é quase nula (BEHRENS; WEIGERT; FETZER, 2013).

1.1.2.2 Honra: Um processo defeituoso nunca cria mensagens válidas

Em um ambiente não malicioso a propriedade de honra pode ser garantida através de validações no estado do processo, impedindo que o mesmo envie qualquer mensagem caso seja detectada a corrupção acidental de seus dados devido a problemas de *hardware*, ou devido a desvio na especificação do algoritmo (BEHRENS; WEIGERT; FETZER, 2013).

1.2 Paxos

Dentre os algoritmos de consenso mais conhecidos atualmente destaca-se o algoritmo de Paxos, no qual os processos do sistema devem concordar sobre um valor ou operação para que a mesma seja efetivada. Tal escolha se dá por meio de uma votação, onde é ratificada uma decisão que possuir a maioria simples dos votos. Para que uma votação seja válida, é necessária a participação de $\lfloor N/2 \rfloor + 1$ instâncias, onde N é o número total de processos participantes. A essa quantia mínima necessária de participantes é dado o nome de *quórum*. Além do quórum, também é necessário um *coordenador*, um processo responsável por lançar as propostas e gerenciar as votações. O coordenador é

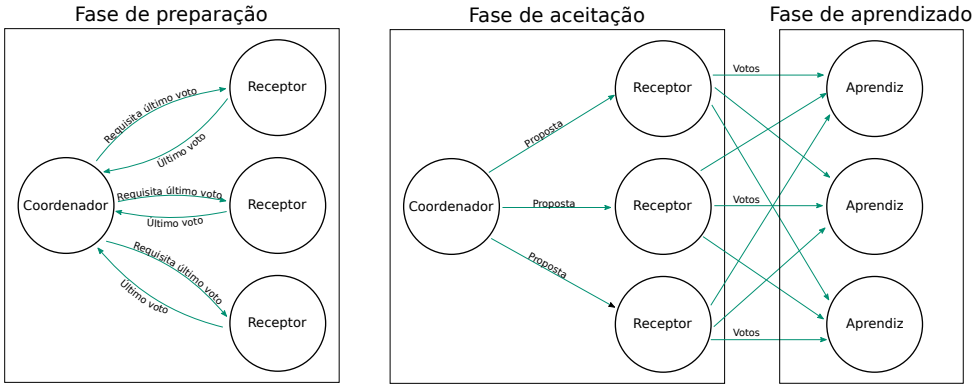
escolhido dentre os N processos participantes através de um algoritmo de eleição de líder (LAMPORT, 1998).

Para fins de notação, dizemos que o Paxos é composto por um processo *coordenador* que emite propostas de votação e as gerencia, enquanto processos capazes de solicitar a emissão de propostas ao coordenador são chamados de *proponentes*. Há ainda os processos *receptores*, responsáveis por votar nas propostas e por fim os *aprendizes*, que recebem a decisão e executam a ação (LAMPORT, 1998).

Cada *rodada de consenso* do Paxos possui três fases, as quais foram ilustradas na Figura 1. A primeira delas, chamada de *fase de preparação*, se inicia com o coordenador enviando uma mensagem para todos os receptores solicitando a participação dos mesmos na votação. Os receptores que aceitarem a requisição respondem ao coordenador enviando uma mensagem que contém o valor do último voto que efetuaram. Na segunda fase, chamada de *fase de aceitação*, o coordenador verifica se recebeu repostas de receptores o suficiente para realizar uma votação, ou seja, um quórum. Em caso positivo, o coordenador escolherá entre as respostas obtidas o valor com maior número de rodada para ser decidido, e o enviará em forma de proposta aos receptores para que votem. Os receptores então recebem a proposta e transmitem seu voto aos aprendizes. Por fim, na *fase de aprendizado* os aprendizes aprendem o resultado da proposta ao receberem mensagens de um quórum de receptores com a mesma decisão. Este fluxo constitui uma rodada de consenso, e em casos onde falhas ocorrem durante o fluxo, várias rodadas podem ser necessárias até que a proposta seja decidida (LAMPORT, 1998).

O processo de decisão de uma proposta, incluindo todas rodadas de consenso que forem necessárias até que as réplicas cheguem em um consenso, é classificado como uma *instância* do algoritmo Paxos, e para cada proposta o Paxos criará uma instância (LAMPORT, 1998). Em uma aplicação prática, a sequência de propostas na ordem em que as instâncias são executadas pode ser estruturado no modelo de máquinas de estados deterministas (SCHNEIDER, 1990).

Figura 1: Rodada do algoritmo Paxos

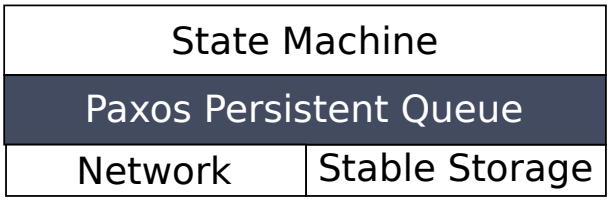


1.3 Treplica

Para que todo potencial da redundância e replicação possa ser aproveitado é necessário que certos obstáculos sejam superados, como a disponibilidade apesar da falha de certos componentes, desempenho e simplicidade de programação (VIEIRA; BUZATO, 2008; VIEIRA; BUZATO, 2010). Os dois últimos são atendidos pelo Treplica (VIEIRA; BUZATO, 2008; VIEIRA; BUZATO, 2010), um *framework* baseado em Paxos para desenvolvimento de aplicações distribuídas, através da implementação de replicação ativa baseada em consenso e de um modelo de programação simples e orientado à objeto para replicação.

Priorizando a consistência e com o objetivo de aumentar a confiabilidade e desempenho de um sistema distribuído, a replicação ativa é uma técnica que consiste em replicar o estado de um processo entre diversas instâncias, onde cada uma dessas instâncias é chamada de *réplica* e se comporta como uma máquina de estados finita (SCHNEIDER, 1990). Através do Paxos, as transições de estado são executadas em todas máquinas na mesma ordem, garantindo que todas estejam no mesmo estado após executarem o mesmo número de eventos (VIEIRA; BUZATO, 2008; VIEIRA; BUZATO, 2010). A proposta apresentada no Treplica, como apresentado na Figura 2, é composta por dois componentes principais, sendo eles uma *máquina de estados replicada* chamada *State Machine* e uma *fila persistente assíncrona*, chamada *Paxos Persistent Queue*.

Figura 2: Arquitetura do Treplica



A máquina de estados replicada é a camada mais externa do Treplica, tendo contato direto com a aplicação e abstraindo para a mesma as funcionalidades da fila persistente, provendo uma interface que torna transparente para o usuário as funcionalidades de replicação e redundância. Toda transição de estado só é efetuada após ser recebida através da fila persistente, onde ela é confirmada por um quórum de réplicas através de trocas de mensagens utilizando a camada *Network*. Enquanto isso, filas persistentes assíncronas permitem que processos troquem objetos de forma totalmente ordenada. No Treplica, a fila serve como um registro persistente de todos eventos que causam transições de estado, para que tais eventos possam ser reproduzidos caso a máquina de estados perca o seu estado atual devido a uma falha ou pausa na réplica, e é nessa camada da arquitetura que encontra-se a implementação do algoritmo Paxos. Os componentes do algoritmo estão estruturados como classes Java e a nomenclatura adotada é a mesma apresentada por Lamport (2006), como está descrito na Figura 4.

Na arquitetura do Tréplica o coordenador é representado através da classe *Coordinator* e os receptores pela classe *Acceptor*. Para simplificar a implementação, proponentes e aprendizes são representados pela mesma classe, chamada *Learner*. Já que proponentes são responsáveis por receber ações da máquina de estados e enviar em forma de propostas ao coordenador, e aprendizes por receber o resultado das propostas e enviar à máquina de estados, é válido unir tais componentes em uma só classe a fim de centralizar a comunicação entre a máquina de estados e a fila persistente. Além dos componentes inerentes ao algoritmo Paxos o sistema também conta com o componente *Secretary*, responável por organizar a escrita das operações na camada de persistência, abstraída pelo componente *Changelog*, além do *Router* que orquestra a comunicação entre os componentes através de mensagens encapsuladas no componente *Transport*.

Para compreender o trabalho desenvolvido é essencial entender também o papel de dois componentes da arquitetura, mais especificamente o *State Machine* e o *Learner*, que serão detalhados a seguir.

1.3.1 State Machine

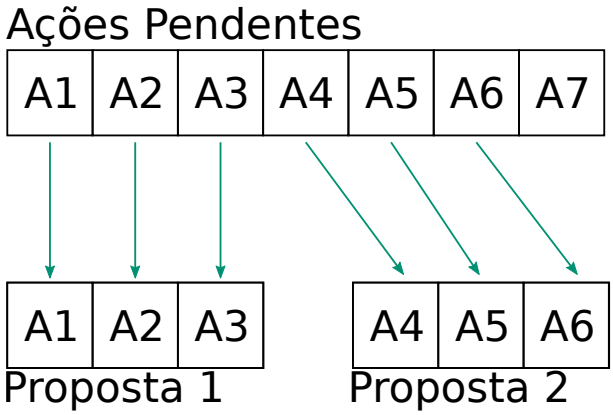
Como já dito, a máquina de estados é a camada que entra em contato com a aplicação cliente e abstrai para a mesma todas as funcionalidades do Paxos. É nela que chegam todas as requisições para efetuar ações, e também é ela que devolve o resultado da ação.

A aplicação solicita à máquina de estados uma mudança de estado através de uma ação. Uma ação é um objeto no qual é encapsulada uma chamada de método junto com seus argumentos, onde a execução de tal método acarreta na transição de estado. Ao receber o pedido de execução dessa ação, o *State Machine* bloqueia a *thread* da aplicação até que a mudança de estado de fato ocorra e adiciona a ação à fila persistente assíncrona onde todo o fluxo do Paxos irá acontecer. Após a proposta na qual a ação foi incluída ser decidida através do Paxos, ela retornará ao *State Machine* para que o método contido na ação seja de fato executado. Após a execução a *thread* da aplicação que havia solicitado a ação é desbloqueada, recebendo o resultado do método contido na ação como valor de retorno. Ao final desse fluxo a máquina de estados passa a ter um novo estado à partir do qual a próxima ação será executada. É importante notar que o bloqueio da *thread* não limita a aplicação à solicitação de uma ação por vez, já que ela pode criar várias *threads* para requisitar ações concorrentemente.

1.3.2 Learner

Após a aplicação requisitar uma ação à máquina de estados, a mesma adiciona tal ação à fila persistente para que seja decidida através do algoritmo Paxos. À partir

Figura 3: Transformação das ações em propostas

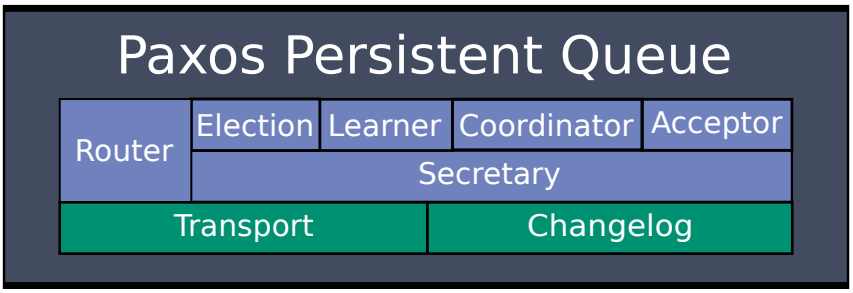


daí, é responsabilidade do *Learner* inseri-la em uma proposta e enviá-la ao coordenador requisitando sua decisão. O processo de adicionar ações a uma proposta segue algumas regras que podem ser vistas na Figura 3, a fim de otimizar o tempo de decisão do Paxos. Primeiramente, como uma decisão pode levar um tempo considerável (dezenas de milissegundos), durante esse tempo várias ações podem ser requisitadas por várias *threads* do cliente. Assim, dentro de uma proposta são encapsuladas diversas ações pendentes, de forma ordenada, até atingir um tamanho máximo parametrizado para a proposta, a fim de que elas sejam decididas de uma só vez e entregues para serem executadas na máquina de estados. Além disso, duas propostas podem estar em processo de decisão ao mesmo tempo, adicionando um fator de paralelismo ao Paxos.

O *Learner*, por concentrar os papéis de proponente e aprendiz, é essencial para o início e fim de uma rodada do algoritmo Paxos. O início de uma rodada do procolo se dá através do *Learner* quando ele, cumprindo o papel de proponente, envia através da rede, fazendo uso do componente *Transport*, uma mensagem ao *Coordinator* contendo uma proposta que deve ser decidida. O coordenador por sua vez ordena a proposta junto às demais recebidas de outros proponentes e a transmite através de uma mensagem para todos os *Acceptors* a fim de que ela seja votada pelos mesmos. Os *Acceptors* transmitem seus votos através de uma mensagem de *broadcast*, ou seja, uma mensagem enviada para todos os processos do sistema. Esta mensagem inclui um identificador do seu autor, um identificador do processo de origem da proposta, a rodada de decisão à qual o voto do *Acceptor* se refere e o identificador da proposta que o mesmo está tentando decidir. Por fim, o *Learner* ao receber tais mensagens através da rede, agora cumprindo seu papel de aprendiz, verifica se há um quórum de votos para a proposta em questão. Em caso positivo, ele extrai as ações de dentro da proposta de forma ordenada e as envia ao *State Machine* para que sejam executadas.

Além do envio à máquina de estados, o *Learner* também faz uso do componente *Changelog* para escrever as ações em disco em forma de *log*, a fim de sejam reexecutadas caso a instância em questão seja parada e iniciada novamente.

Figura 4: Fila Persistente Assíncrona



1.3.3 Transport e Changelog

Os componentes *Transport* e *Changelog* são responsáveis por auxiliar os demais componentes em atividades que envolvem, respectivamente, interações com a rede e o disco, a fim de isolar responsabilidades e manter as tarefas centralizadas.

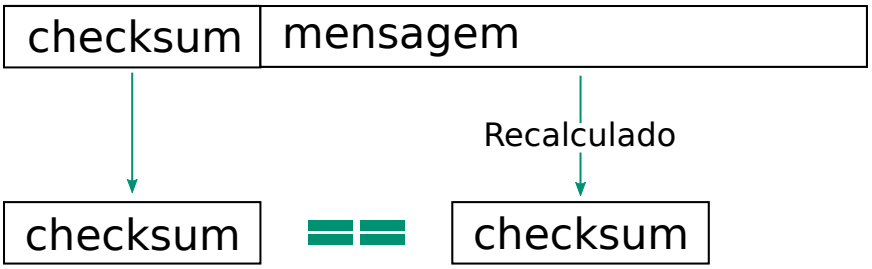
Através do *Transport* é que todas mensagens necessárias para o funcionamento do Paxos são enviadas e recebidas, se comunicando diretamente com a rede. Todas mensagens enviadas e recebidas por esse componente são encapsuladas em um objeto padronizado contendo o identificador do processo de origem, o tipo de mensagem (voto, proposta, etc.) e o seu conteúdo.

O componente *Changelog* é responsável pela comunicação do Treplica com o disco, ou seja, pela escrita e leitura dos dados persistentes. Todas transições de estados são escritas no disco através do *Changelog*, mantendo assim um histórico da máquina de estados em forma de arquivo, para que essas ações possam ser reexecutada quando necessário. O cenário onde essa execução é necessária é quando a réplica em questão tem seu processamento interrompido por algum motivo, por exemplo uma queda de energia, e quer se reintegrar ao ambiente. Para que o processo nessa nova execução tenha o mínimo de perda possível do histórico de ações, é realizada a leitura do arquivo de *log* através do *Changelog* a fim de recuperar sua máquina de estados até o último estado em que ele esteve operante, diminuindo assim a lacuna de propostas das quais ele não tem conhecimento (pois encontrava-se inoperante) e que precisarão ser requisitadas às demais réplicas, já que esse processo é mais custoso do que a reexecução do *log*.

1.4 Paxos arbitrário não malicioso

A biblioteca Treplica originalmente, devido às características do algoritmo Paxos, é tolerante a falhas no modelo *fail-recovery*. No entanto o trabalho de Barbieri (2016) apresenta uma extensão da biblioteca para o modelo de tolerância a falhas não maliciosas, criando validações de consistência em diversos pontos da arquitetura. Esses pontos críticos são os locais onde as falhas que não estão no modelo *fail-recovery* são identificadas, fazendo com que fossem necessárias uma série de categorias de validações diferentes para uma

Figura 5: Validação de integridade



tolerância abrangente. À seguir serão descritas as validações implementadas no trabalho citado, assim como em que ponto do Treplica cada uma se encontra.

1.4.1 Validação de Integridade

A validação de integridade é a primeira medida para a detecção de falhas no sistema, e consiste na criação de um *checksum* (BHATOTIA et al., 2010) do conteúdo de cada mensagem Paxos assim que a mesma é instanciada, seja para ser salva no disco ou propagada na rede. Toda vez que uma mensagem é recuperada do disco ou recebida através da rede por uma das réplicas, o *checksum* é recalculado à partir de seu conteúdo e validado contra o original anexado à mensagem, como mostra a Figura 5. A camada de rede já utiliza de forma nativa um método semelhante para validar mensagens recebidas, porém foi decidido por realizar a validação novamente de forma explícita para manter um padrão no formato das mensagens trocadas através da rede e aquelas recuperadas do disco. Através desse método é possível identificar falhas na rede ou disco que sejam capazes de corromper mensagens (BARBIERI; VIEIRA, 2015; BARBIERI, 2016).

Dentro da estrutura do Treplica é possível identificar a validação de integridade em todos componentes que recebem mensagens, sejam elas enviadas por outro componente através do *Transport* ou durante a reexecução de eventos salvos no *Changelog*. No componente *Coordinator*, por exemplo, todas propostas recebidas provenientes dos *Learners* são validadas antes que possam ser enviadas para votação, enquanto o *Learner* valida todo voto recebido por um *Acceptor* antes de processá-lo. No *State Machine* ao receber ações do *Learner*, antes de executá-las, a validação de integridade também ocorre. No componente *Secretary*, o momento de gravar um novo *checkpoint* no disco, também é executada a validação de integridade, e por fim no *Changelog*, quando é necessário reexecutar os eventos do *log* em uma instância, cada evento tem seu *checksum* validado antes de ser executado.

1.4.2 Validação de Estado

Inspirado no Git[1] em que todo *commit* possui um *hash*, para esta validação é gerado um *checksum* para cada estado da máquina de estados. Um novo *checksum* é

[1] Documentação disponível em: <https://git-scm.com/>

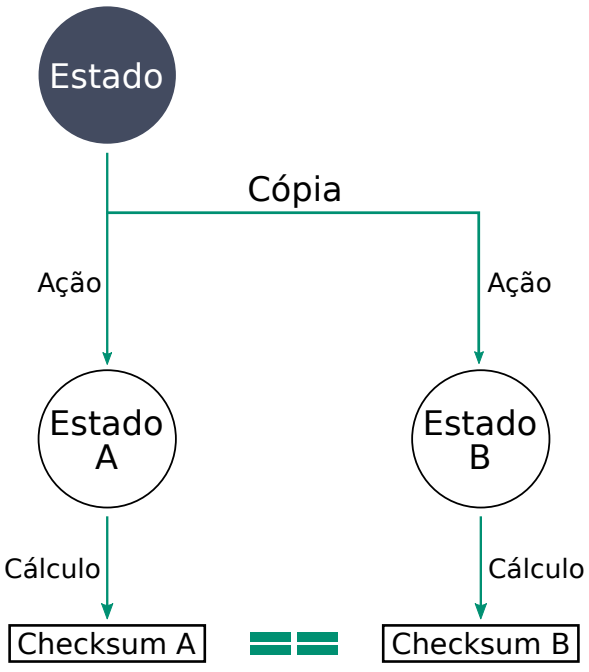
calculado cada vez que há uma transição de estados, e o mesmo é recalculado com base no *checksum* do estado anterior, nos dados que acarretaram na mudança de estado e nas informações do estado atual (BARBIERI; VIEIRA, 2015; BARBIERI, 2016).

Calcular o *checksum* utilizando todos os dados do estado, dependendo do tamanho do mesmo, pode ser muito custoso e desnecessário. Pensando nisso o sistema oferece duas opções para realizar tal cálculo. A primeira delas, chamada de *deep check*, consiste basicamente em utilizar todo o dado do estado para o cálculo. É uma solução completa e que pode ser vantajosa para casos em que as informações não cresçam indefinidamente. Em outros casos onde esse procedimento pode se tornar muito custoso outra abordagem pode ser utilizada, na qual fica sob responsabilidade da aplicação fornecer um conjunto de dados representativo do estado (BARBIERI; VIEIRA, 2015; BARBIERI, 2016).

Para validar um estado, antes de cada transição, é criada uma cópia do estado atual, onde todas as ações da proposta decidida são efetuadas de forma idêntica ao original e sobre a qual também é realizado o cálculo do *checksum* após cada mudança. Por fim, como na Figura 6, toda vez que o estado é requisitado é comparado seu *checksum* com o de sua cópia, a fim de prevenir possíveis corrupções de memória que possam fazer um estado divergir (BARBIERI; VIEIRA, 2015; BARBIERI, 2016).

Validações de estado ocorrem especificamente em duas situações, ambas dentro do componente *State Machine*. A primeira delas após a execução de uma ação recebida pelo *Learner* e, depois disso, toda vez que o estado atual é requisitado, seja na lógica interna ou na aplicação cliente.

Figura 6: Validação de estado

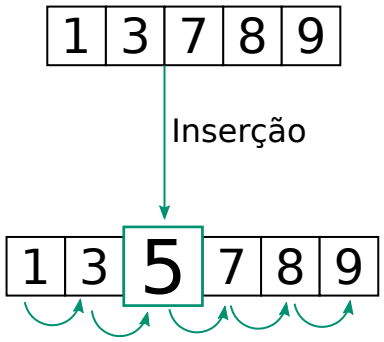


1.4.3 Validação Semântica

A fim de detectar falhas em memória principal ou no código da aplicação, é possível realizar validações semânticas sobre as transições de estado. Nessa categoria de validação fica a cargo da aplicação fornecer, para cada transição, um método capaz de verificar se o resultado final da transição condiz com o esperado, e o mesmo é executado após cada ação na máquina de estados. (BARBIERI; VIEIRA, 2015; BARBIERI, 2016).

Na Figura 7 é proposta uma aplicação que implementa uma lista ordenada utilizando Treplica. Nesse caso, uma possível validação semântica seria checar se, após uma ação de inserção ou remoção, a lista permanece ordenada. Tal validação ocorre no componente *State Machine*, após a execução de uma ação recebida pelo *Learner*, nos dois estados gerados durante a Validação de Estado detalhada na Seção 1.4.2, para certificar-se de que o novo estado da máquina de estados é válido de acordo com a semântica da aplicação.

Figura 7: Validação semântica

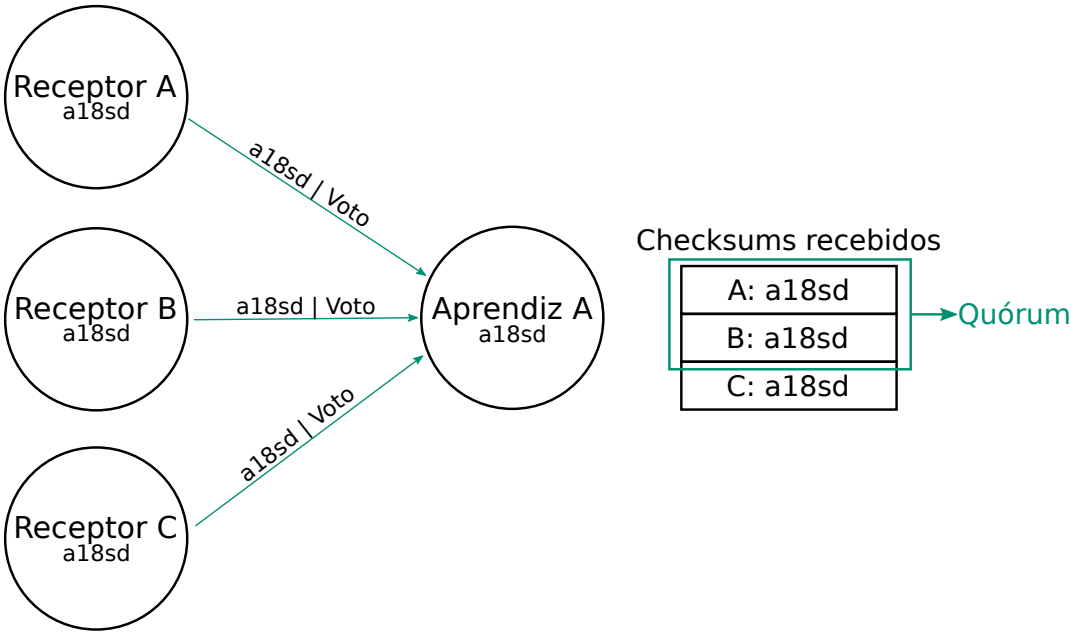


1.4.4 Validação Distribuída

Para que seja possível tolerar falhas arbitrárias não é suficiente apenas validar se o estado de uma réplica não corrompeu-se durante o processamento, mas também se o mesmo não diverge se comparado ao estado equivalente nas demais réplicas. Sendo assim o foco desta validação não é detectar problemas causados por corrução de estado, mas sim por erros de código, configuração de sistema ou desvios do modelo computacional no ambiente. Estes desvios não são identificados quando analisadas as réplicas de forma independente, mas constituem em uma divergência do comportamento correto quando analisado o sistema como um todo. Para que seja possível realizar tal validação, houve uma extensão do algoritmo Paxos originalmente utilizado no Treplica para anexar à mensagem de voto enviada pelos *Acceptors* de cada instância o *checksum* do seu estado atual (BARBIERI; VIEIRA, 2015; BARBIERI, 2016).

O *checksum* utilizado para as validações é gerado através do estado resultante da execução de uma ação. No entanto, a Validação Distribuída ocorre no *Paxos Persistent Queue* e visa a validação de propostas, já que nessa camada as ações não existem

Figura 8: Validação distribuída



independentemente, apenas agrupadas nesses objetos. Sendo assim uma proposta gera diversos *checksums* diferentes, um para cada ação contida nela, o que se torna um problema para a comparação entre as instâncias. Suponha que dois processos A e B decidiram a mesma proposta sem que ocorressem problemas, no entanto a máquina de estados do primeiro processo já executou 4 ações contidas em tal proposta enquanto o segundo só executou duas. Caso os *checksums* que são anexados ao voto fossem atualizados a cada ação executada, os dois processos embora corretos apresentariam nesse momento dois *checksums* de estado diferentes para a mesma proposta.

Para contornar esse problema foi criada uma janela de atualização do *checksum* de estado que é anexado ao voto. A atualização só ocorre a cada N ações, onde N é um número parametrizado que deve ser maior do que o número de ações que cabem em uma proposta. Até que esse número de transições de estado ocorram, o mesmo *checksum* é enviado e utilizado para realizar todas as validações, a fim de promover a sincronização entre as réplicas e viabilizar a comparação (BARBIERI; VIEIRA, 2015; BARBIERI, 2016).

Quando o componente *Learner* de uma réplica recebe o voto dos *Acceptors* das demais instâncias, semelhante ao cenário apresentado na Figura 8, é procurado um quórum de *checksums* idênticos e, caso encontrado, esse é considerado o valor "correto" para o estado sendo validado. Se o estado da réplica que está realizando a validação diverge do quórum ela é considerada defeituosa e imediatamente deixa de operar. Esse método considera improvável o cenário em que a maioria das réplicas divirjam ao mesmo tempo para o mesmo estado quando considerado um sistema composto por mais de 3 instâncias.

2 Validação Distribuída

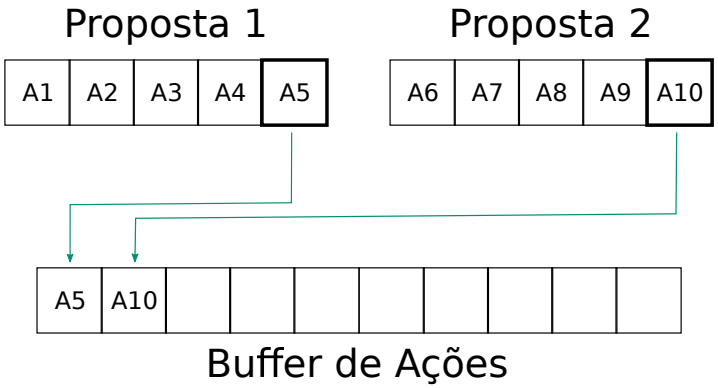
As validações criadas por Barbieri (2016) trouxeram avanços ao Treplica no que diz respeito a tolerância a falhas no modelo arbitrário não malicioso, no entanto, através da análise do trabalho é possível notar que o modelo ainda não é totalmente respeitado pelo algoritmo, o que é evidenciado através dos resultados dos experimentos do trabalho. Como explicado na Seção 1.4.4 a Validação Distribuída pode interromper o funcionamento de um processo se perceber que o mesmo está corrompido, e essa verificação ocorre através do transporte do *checksum* de estado na mensagem de voto enviada do *Acceptor* para o *Learner*. No entanto a validação não ocorre de forma instantânea, ou seja, até que um processo defeituoso seja identificado o mesmo já pode ter enviado estados incorretos ao cliente. Este envio vai contra a propriedade de honra do modelo apresentada na Seção 1.1.2.2, já que o processo defeituoso enviará diversas mensagens válidas até que seja detectado, o que também aumenta as chances do erro ser propagado para as demais réplicas. Além disso, caso dentro de uma janela de validação um quórum divirja entre si, a validação permitirá que o algoritmo progrida mesmo que claramente o sistema tenha tornado-se defeituoso. Suponha um ambiente com 5 réplicas, logo com quórum 3, onde na mesma janela de validação os processos A, B e C divergem respectivamente para os estados X, Y e Z. Nesse caso, independente dos estados dos demais processos não será possível formar um quórum de *checksums* para nenhum dos estados, impedindo a validação de detectar a falha e permitindo o progresso do algoritmo, inclusive o envio de tais estados divergentes para os clientes. O envio de estados incorretos para a aplicação cliente é extremamente problemático em um cenário prático e a reversão dessas operações após serem consolidadas é uma ação complexa.

A fim de solucionar tais problemas e garantir as propriedades fundamentais do modelo de falhas arbitrárias não maliciosas esta dissertação apresenta um nova validação distribuída, invertendo de certa forma a ideia original. Na nova validação, ao invés de um processo ser interrompido apenas quando é detectada uma corrupção em seu estado através de um quórum, toda máquina de estados após avançar para um novo estado, mas antes de enviar o mesmo à aplicação cliente, espera até que o estado seja validado por um quórum, evitando o envio de estados corrompidos à camada cliente.

2.1 Nova Validação Distribuída

A validação desenvolvida tem início no momento em que uma proposta é decidida pelo *Learner*. Como descrito na Seção 1.3.2, uma proposta é composta por diversas ações ordenadas, e antes de enviar tais ações à máquina de estados um identificador da

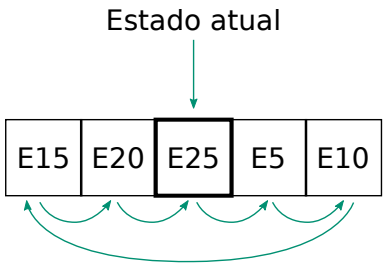
Figura 9: Buffer de Ações



última delas é registrado em um *buffer de ações*, com um comportamento semelhante ao apresentado na Figura 9. A última ação de uma proposta representará a proposta como um todo durante a validação. Devido à forma como o *checksum* do estado é criado (Seção 1.4.2) é possível afirmar que, se um estado está correto, todos estados anteriores também estão corretos, onde estar correto significa ter um *checksum* idêntico ao de um quórum de réplicas.

O componente *State Machine* ao receber as ações, se a *thread* da máquina de estados não estiver em espera como será explicado, imediatamente começa a executá-las uma a uma. Todos estados "intermediários", ou seja, estados gerados à partir da primeira até a penúltima ação de uma proposta, passam por todas validações descritas na Seção 1.4, enquanto o último estado além de passar por todas validações também será o representante da proposta na validação distribuída. Assim que é detectado que o identificador da ação está presente no *buffer*, o identificador é retirado da estrutura e o *checksum* do estado gerado à partir da ação é armazenado em uma *lista circular de validações* como a da Figura 10, ocupando a posição do estado mais antigo da lista, que será usada na validação distribuída. Até que tal estado seja validado, a *thread* do *State Machine* é congelada, evitando assim que ela envie à camada cliente dados incorretos.

Figura 10: Lista Circular de Validações de Estados

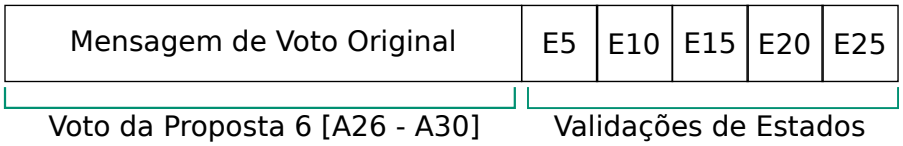


A lista circular de validações de estado foi adicionada à mensagem de voto dos *Acceptors* descrita na Seção 1.3.2 a fim de permitir a validação não só do estado atual como de estados anteriores, para o caso de existirem réplicas atrasadas devido à característica assíncrona do algoritmo. Assim que uma proposta é decidida e suas ações executadas ela é

adicionada à lista circular, logo um voto referente à Proposta 6 possivelmente possuirá em sua lista circular as validações de estado das propostas de 1 a 5, situação ilustrada na Figura 11. Semelhante à validação distribuída descrita na Seção 1.4.4, os *Learners* procurarão por um quórum de *checksums* iguais ao do seu estado atual. No entanto, a busca não é por uma corrupção do estado para interrupção do processo, e sim por uma confirmação de que o estado é válido para que o processo possa continuar.

Quando o *Learner* de uma réplica recebe um novo voto, ele primeiramente irá processar a lista de validações recebidas. Validações de estados passados em relação ao estado da réplica que recebeu o voto serão descartadas, as demais serão armazenadas em um *pool de validações*. Após esse armazenamento, é checado no *pool* se, para o estado atual da réplica, há um quórum de *checksums* advindos de diferentes réplicas. Ao encontrar um quórum de *checksums*, caso sejam iguais ao seu, o *Learner* envia um sinal ao *State Machine* para que o mesmo dê continuidade ao seu processamento, caso contrário a réplica tem seu processo encerrado em definitivo.

Figura 11: Lista circular de validações de estados como *piggyback* de um voto



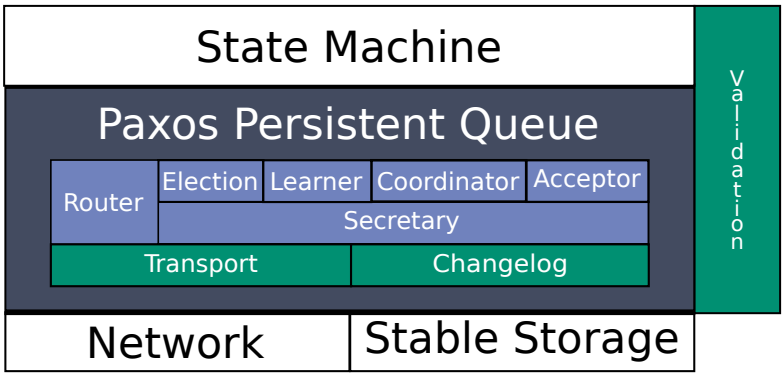
Ao receber uma notificação do *Learner* para continuar seu processamento, o *State Machine* termina de processar o estado que estava sendo validado, finalizando as validações descritas na Seção 1.4 e enviando à aplicação cliente. A partir daí, a máquina de estados está pronta para processar as ações da próxima proposta que estiver empilhada, caso haja alguma, e todo o ciclo se repete.

Em momentos de ociosidade do sistema, ou seja, em que não há rodadas de consenso acontecendo, o *State Machine* pode permanecer com seu processo parado por tempo indeterminado até que novos votos ocorram entre as réplicas para finalizar uma validação. Para que isso não ocorra, um tipo especial de mensagem foi criada para ser enviada após um período predeterminado sem que uma votação ocorra. Tal mensagem não possui um conteúdo significativo, carregando apenas a lista de validações como *piggyback* para ser utilizada na validação distribuída.

2.2 Alterações no projeto original

Para que o novo método de validação proposto fosse possível, alguns pontos do projeto precisaram ser alterados, assim como novos componentes foram criados. Tais modificações serão descritas em detalhe a seguir.

Figura 12: Arquitetura Treplica com a Camada de Validação



2.2.1 Camada de validação

Para a realização do novo método de validação uma comunicação intensa entre os componentes *State Machine* e *Paxos Persistent Queue* foi estabelecida. Essa comunicação é necessária principalmente porque cada uma dessas camadas tem uma visão diferente dos estados, e a combinação de ambas é necessária para implementar todos os passos da validação.

No *Paxos Persistent Queue*, mais precisamente no *Learner*, as ações são "empacotadas" em propostas e essa visão é importante para que seja identificada a última ação de cada proposta, o que permite a criação das janelas de validação dinâmicas. Também é no *Learner* que o *Paxos Persistent Queue* recebe os votos dos *Acceptors*, esses votos possuem o *piggyback* das validações de estado de cada réplica que serão usadas para a validação distribuída do estado atual.

Já no *State Machine* as ações são executadas de forma individualizada, tendo acesso ao seu conteúdo e realizando as mudanças na máquina de estados. Esse nível é necessário para criar os *checksums* dos estados e armazenar o último *checksum* de cada proposta, que será usado pelo *Learner* na validação distribuída. A fim de centralizar essa comunicação em um único ponto foi criado um novo componente na arquitetura original do Treplica chamado de *Validation*. Esse componente, além de servir como um "mensageiro" entre o *State Machine* e o *Paxos Persistent Queue*, atravessando essas duas camadas, é nele que se encontram todas estruturas criadas para a realização da validação distribuída, como o buffer de ações e a lista circular de validações de estado. A nova arquitetura completa contando com o novo componente pode ser vista na Figura 12

2.2.2 Buffer de ações a serem validadas

Em um ambiente com fluxo intenso de ações sendo enviadas pelo cliente haverá um acúmulo de propostas cujos estados devem ser validados. Para um estado ser validado em uma réplica é necessário que essa receba um quórum de *checksums* para aquele estado iguais ao que ela possui. O processo de criação, espera e comparação do *checksum* com as

demais réplicas leva um tempo maior para ocorrer do que a decisão de uma proposta pelo Paxos, o que justifica o acúmulo de propostas já decididas mas ainda não validadas.

Com o intuito de suportar tal acúmulo sem deixar nenhuma proposta ser entregue sem validação, um *buffer* foi criado na camada de validação para registrar a última ação de todas as propostas já decididas que ainda não foram validadas, como exibido na Figura 9. Assim que chega o momento correto de validar tal proposta, ou seja, todas propostas anteriores já foram validadas, ela é retirada do *buffer* e seu *checksum* é gerado e inserido na lista circular de validações de estado para que seja validado e enviado como *piggyback* para outras réplicas.

2.2.3 Janela de validação dinâmica

Na implementação proposta por Barbieri (2016) as janelas de validação possuíam tamanho fixo. Tendo o conhecimento aproximado do fluxo de operações que serão realizadas no ambiente tal valor era configurado para que fosse próximo do tamanho ideal. O tamanho ideal para uma janela de validação é exatamente o número de ações dentro de uma proposta, de forma que sempre a última ação de cada proposta seja validada. Uma janela de validação maior do que o número de ações em uma proposta pode fazer com que estados ou até mesmo propostas inteiras corrompidas só sejam detectadas na consolidação da proposta seguinte, ou seja, quando o erro já se propagou, enquanto uma janela menor que o tamanho da proposta fará com que ocorram diversas validações dentro de uma mesma proposta, ação desnecessária devido à forma como os *checksums* são construídos e que apenas adiciona custo computacional ao sistema. Outro ponto que precisa ser salientado é que o tamanho das propostas é variável. Propostas são enviadas quando chegam a um tamanho máximo ou um *timeout* ocorre, logo em momentos de baixo fluxo de ações as propostas possuirão menos ações do que em momentos de alto fluxo. Dado todos esses fatores, tornou-se evidente a importância de um controle maior sobre o tamanho das janelas de validações, de forma que fossem precisas e variassem de acordo com o tamanho das propostas.

A maior dificuldade para criar janelas de validação precisas se dá pelo fato do *State Machine*, onde a janela é criada, não ter visão das propostas, apenas das ações. A criação da camada de validação gerando uma visão compartilhada entre o *State Machine* e o *Paxos Persistent Queue* permite que a informação do tamanho de cada proposta recebida, informação essa extraída pelo *Learner*, possa ser utilizada na máquina de estados para a criação de janelas dinâmicas. Para que isso seja possível dois contadores de ações foram criados no componente *Validation*, sendo um incrementado na fila persistente e outro na máquina de estados. No *Paxos Persistent Queue* o contador incrementa antes de cada proposta ser entregue à máquina de estados de acordo com o número de ações que constituem a proposta enquanto no *State Machine* o contador é incrementado a cada nova ação executada. Quando ambos os contadores possuem o mesmo valor significa que a

ação em questão é a última da proposta entregue, e então o *checksum* do estado gerado é armazenado para ser usado na janela de validação atual. Dessa forma possuímos uma janela de validação que varia de tamanho a cada proposta decidida, garantindo que nenhuma proposta seja entregue à camada cliente sem ser validada ou então que ela seja validada diversas vezes de forma desnecessária.

2.2.4 Lista circular de validações de estado

Na implementação proposta por Barbieri (2016) cada réplica envia como *piggyback* apenas seu estado atual. No entanto, com o novo conceito de validação uma réplica só avança seu estado quando existir um quórum de réplicas com o mesmo *checksum* para o estado atual. Sendo assim, uma réplica que se encontra atrasada em relação às demais nunca avançará seu estado, pois não consegue validar o mesmo. Para resolver esse problema uma lista circular com os *checksums* dos últimos N estados passou a ser armazenada e enviada como *piggyback*, onde N é um valor parametrizável. Dessa forma, réplicas atrasadas em até N estados conseguem realizar sua validação sem problemas, enquanto réplicas demasiadamente atrasadas são permitidas de avançar seu estado sem validação, até que o deficit de estados torne-se menor ou igual a N. Para os testes realizados foi utilizado o valor de N igual a 5, como na Figura 10, pois após observações foi notado que enquanto o Paxos opera em regime as réplicas tendem a avançar seus estados no mesmo ritmo, tendendo a não ficar mais do que 5 estados para trás.

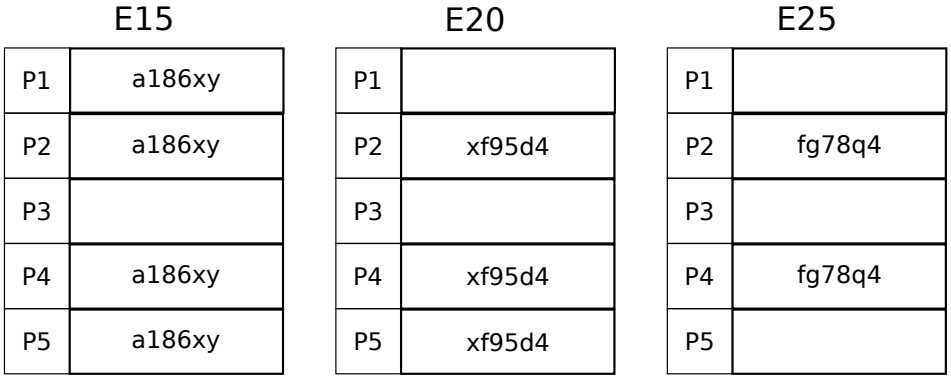
2.2.5 Piggyback da lista circular de validações de estado

Na validação distribuída as réplicas precisam enviar novos tipos de mensagens às demais para compartilhar a lista contendo suas últimas validações de estado. Para não aumentar consideravelmente o volume de troca de mensagens entre as réplicas, foi decidido enviar tal lista como *piggyback*, ou seja, como um conteúdo extra em um tipo de mensagem já existente no algoritmo Paxos, mais especificamente a mensagem de voto, realizando a validação a cada voto recebido. Como um nó entrega uma proposta para o *State Machine* logo após receber um voto que forma um quórum, é garantido dessa forma que um nó não entregará uma proposta ao *State Machine* caso não passe pela validação distribuída para o seu estado atual. Como é possível ver na Figura 11, as validações da lista circular de validações são ordenadas de acordo com a ordem dos estados que elas representam para serem anexadas à mensagem.

2.2.6 Pool de validações

O *pool* de validações consiste em uma estrutura na qual um nó armazena as validações de estados futuros em relação ao seu estado atual recebidas de outros nós.

Figura 13: *Pool* de validações



Basicamente, para cada estado é criada uma tabela dentro do *pool*, que será preenchida com os *checksums* fornecidos pelos demais processos através de trocas de mensagem. As tabelas são criadas e preenchidas conforme as mensagens de votos dos demais processos são recebidas, o que significa que processos atrasados ao receber *checksums* de estados nos quais eles ainda não chegaram armazenam esses *checksums* para utilizar na validação quando chegar o momento correto. Dessa forma uma réplica pode validar seu novo estado instantaneamente se já possuir um quórum para o mesmo, advindo de um quórum de nós adiantados em relação a ela. Esse *pool* também torna mais raro o caso em que uma réplica está tão atrasada em relação às demais que não possui forma de validar seu estado (ou seja, está mais do que N estados atrasado, sendo N o tamanho da lista circular de validações de estado) decidindo propostas sem que ocorra a validação dos estados anteriores. Na Figura 13 é possível ver como o *pool* funciona. Para cada estado, uma tabela com o *checksum* de validação para ele recebido de cada réplica é armazenado, a fim de buscar um quórum de *checksums* semelhantes para poder seguir no processamento.

2.2.7 Mensagens de Timeout

Todas alterações até então propostas visam viabilizar a validação dos estados enquanto o algoritmo Paxos estiver em regime, ou seja, tudo acontece como o esperado em um fluxo contínuo, com novas propostas sendo decididas e consequentes ações sendo executadas nas máquinas de estados. No entanto, em certos momentos o algoritmo não apresenta tal comportamento ideal, e para contornar tais situações algumas mensagens de *timeout* precisaram ser criadas para permitir o acontecimento da validação. Esse é o caso das mensagens de *timeout de piggyback* e *timeout de validação*, descritos nos parágrafos a seguir, respectivamente.

Em momentos nos quais não há votações acontecendo, um *timeout* é gerado e um tipo de mensagem específico é enviado apenas com as listas de validações como *piggyback*, para que a validação distribuída das propostas pendentes seja finalizada mesmo em momentos de baixo fluxo no ambiente.

Na implementação do algoritmo Paxos proposta por Vieira e Buzato (2008) uma réplica não avança seu estado se ela estiver satisfeita com o estado atual, por exemplo, se nada novo está sendo proposto ou se ela entra em colapso e se recupera. Devido a essa característica, em momentos de ociosidade do sistema a última réplica a propor operações pode encontrar-se em uma situação em que ela não consegue validar seu estado, já que nenhuma outra réplica do ambiente possui o mesmo estado para enviar como *piggyback*. Como medida para prevenir que essa situação ocorra foi criado um *timeout* que, quando acionado, faz com que a réplica que estiver nessa situação force os demais nós do ambiente a atualizarem seu estado para o estado que precisa ser atualmente validado. Essa medida em combinação com o *timeout* de *piggyback* garantem que a validação continue ocorrendo em momentos de baixo fluxo de operações.

3 Experimentos e Resultados

Para que fosse possível validar a efetividade das modificações propostas uma bateria de experimentos foi realizada, com foco principal em avaliar o comportamento do algoritmo em momentos de falha. Além disso, testes de desempenho também foram executados, para que fosse possível medir o desempenho do algoritmo quando o mesmo está em regime, ou seja, enquanto tudo está funcionando como esperado e há um fluxo continuo de ações sendo realizadas.

Primeiramente foi necessário decidir quais tipos de falhas seriam exploradas. Era importante validar que independente do componente do algoritmo Paxos em que ocorresse a falha o sistema a toleraria e, levando isso em consideração, foi decidido injetar 4 categorias de falhas. As 3 primeiras são os alvos da avaliação e ocorrem em cada um dos componentes vitais do algoritmo (*Acceptor*, *Coordinator* e *Learner*) enquanto a última ocorre na comunicação entre réplicas. A falha de comunicação serviu como uma ativadora para as demais falhas, já que sem ela as demais não causam efeitos adversos no algoritmo Paxos original em regime, como será explicado à frente.

Para cada categoria de falha alvo foi criada uma máquina virtual com as mesmas especificações técnicas e configurado um ambiente com diversos componentes. O primeiro deles é o Tréplica em si, compilado de forma especial para incluir o *framework* de injeção de falhas. Com o Treplica devidamente configurado, foi necessário utilizar um gerador de carga para executar uma aplicação usando o Treplica e gerar ações automaticamente, configurando o número de operações por segundo desejadas e o tempo de execução através de configurações parametrizáveis.

O conjunto de gerador de carga e Treplica com injeção de falhas já é o suficiente para a execução de um teste, no entanto para aumentar a confiabilidade dos resultados foi decidido executar 50 testes em sequência para cada categoria de falha, utilizando os mesmos parâmetros, e então extrair a média dos resultados. Para esta execução em massa, foi desenvolvido um *script* responsável por orquestrar tais execuções sequenciais e atribuindo sempre os mesmos parâmetros para o gerador de carga e o Treplica.

Por fim, os resultados das execuções são registrados em diversos arquivos que, por se tratar de diversas execuções e operações realizadas, tornam-se extensos e de difícil manuseio. Sendo assim, ao final das execuções cada bateria de testes foi analisada através de um *parser* capaz de interpretar os resultados registrados e extrair as métricas necessárias para avaliar o algoritmo.

Além das baterias de testes de falha também houve uma bateria de testes de desempenho constituída dos mesmos componentes já citados, sendo a única diferença a

Figura 14: Exemplo de método original

```
private ArrayList<Decree> decrees;

// Método original
public Vote getPrevVote(long decreeNumber) {
    Decree decree = decrees.get(decreeNumber);
    Vote vote = decree.prevVote;
    return vote;
}
```

Figura 15: Exemplo de método com falha

```
// Método com falha
public getPrevVote(long decreeNumber) {
    return null;
}
```

compilação do Treplica realizada de forma tradicional, já que nos testes de desempenho não foram injetadas falhas durante a execução, pois o intuito era medir o desempenho do algoritmo em regime, ou seja, o impacto das validações realizadas quando o fluxo do algoritmo está dentro do esperado.

O restante do capítulo detalhará o funcionamento dos componentes aqui apresentados, os teste realizados, além dos resultados obtidos e o seu significado para o trabalho.

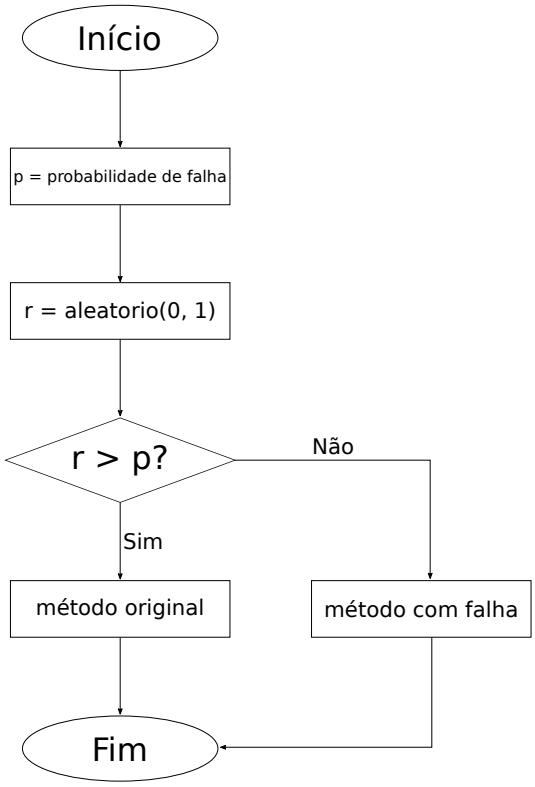
3.1 Framework de Injeção

O *framework* utilizado trata-se de um aprimoramento da ferramenta apresentada por Barbieri (2016) que faz uso do AspectJ[1], uma ferramenta que permite inserir *pointcuts* no código para injetar falhas em tempo de execução dada uma probabilidade para que a falha ocorra. Na prática, é criada uma cópia do método que realizará a ação desejada substituindo o código original, por exemplo o da Figura 14, por outro incorreto, por exemplo, que simplesmente possui um valor de retorno nulo, como na Figura 15. Então, ao invés de executar diretamente uma das versões do método no fluxo do código, é executado um terceiro método que, através dos parâmetros especificados, decide qual versão será executada. Como é possível notar na Figura 16, a condição para que a injeção de falha seja executada é que r seja menor ou igual à probabilidade p parametrizada, onde r é um número aleatório entre 0 e 1.

Para atender melhor aos objetivos dos experimentos a forma como o registro de suas operações é realizado foi modificado. A fim de avaliar as operações de cada instância de cada execução separadamente, foram criados arquivos de registro separados para cada

[1] Documentação disponível em: <https://bit.ly/2XayLb4>

Figura 16: Fluxo de injeção de falha



processo em execução, identificando devidamente seu número de processo e de execução. Por exemplo, o arquivo de registro do processo número 3 da execução número 12 terá o seguinte nome:

```
p3_12.log
```

Dessa forma é possível identificar quais processos tiveram falhas injetadas e quais dessas falhas só foram toleradas graças às alterações propostas no trabalho aqui sendo apresentado.

3.2 Falhas injetadas

Para que fosse possível observar o comportamento do algoritmo Paxos em conjunto com a nova validação em situações complexas e adversas, três componentes essenciais do algoritmo foram escolhidos para terem seu funcionamento deturpado: *Ledger*, *Learner* e *Coordinator*. Tais componentes são fundamentais para todas as rodadas de decisão, logo é esperado que injetar falha em seus métodos gere grandes instabilidades no sistema como um todo, levando a erros em um ambiente que não as tolere corretamente. Além disso, o objetivo dos experimentos era testar falhas que não fossem toleradas pelo modelo de falha e recuperação, como falhas de programação e configuração. Sendo assim, as falhas escolhidas simulam cenários em que o tamanho do quórum, essencial para o funcionamento do Paxos,

foi configurado incorretamente. O intuito é que a validação distribuída tolere todas essas falhas de modo que nenhum erro seja gerado no estado do sistema. Entretanto, como será detalhado, as falhas analisadas por si só não geram erros, para isso elas dependem do acontecimento de uma sequência de outras falhas para desviar o Paxos do seu fluxo padrão. Por exemplo, o reinício de uma rodada de decisão devido à falta de votos, causada por um problema na comunicação entre processos. Por isso, além dessas falhas, em todas as execuções foram injetadas falhas na camada de troca de mensagens, a fim de gerar as sequência necessárias para que os erros ocorram.

Abaixo as falhas testadas serão descritas em detalhe usando como base um cenário hipotético onde elas geram erros nos processos. Para o cenário, será sempre considerado um sistema com 3 processos (A, B e C), logo com quórum de 2 réplicas e com o processo A sendo o coordenador do Paxos.

3.2.1 Falha no payload da mensagem

Como será evidenciado na descrição das demais falhas, elas sempre dependem de um estado adverso do Paxos para de fato gerar um erro, como o coordenador não receber votos suficientes para uma proposta ou a rodada de consenso atual não chegar ao fim para algum *Learner*. Quando o algoritmo Paxos está em regime tais injeções de falhas não geram divergência entre réplicas, pois estão previstas e são toleradas no modelo *fail-recovery* detalhado na Seção 1.1.1.1.

De modo a estimular que rodadas de consenso não se concretizem para que as falhas injetadas tenham efeitos colaterais foi utilizada uma falha na troca de mensagens entre réplicas. Essa falha simplesmente descarta o conteúdo de uma mensagem, que é o equivalente a esse mensagem ter se perdido, tendo um comportamento semelhante ao de instabilidades de rede.

3.2.2 Acceptor esquece seu último voto

A Figura 17 ilustra um cenário onde um erro ocorre ao final após uma sequência de falhas. Para replicar esse cenário, foi criada uma injeção de falha que faz com que, durante a fase de preparação, ao ser solicitado pelo *Coordinator*, o *Acceptor* esqueça qual foi seu último voto e envie um voto nulo no lugar. Além da falha alvo, foi utilizada a falha de mensagem para gerar o cenário em que o processo coordenador não receba votos suficientes para decidir uma proposta.

Em determinada rodada do algoritmo Paxos, durante a fase de aprendizado, a proposta X é decidida nos processos B e C. No entanto, devido à injeção de falhas de mensagens algumas delas se perdem, e o *Learner* do processo A não recebe um quórum de votos para tal proposta, logo para ele a proposta não foi decidida. Como para o processo A

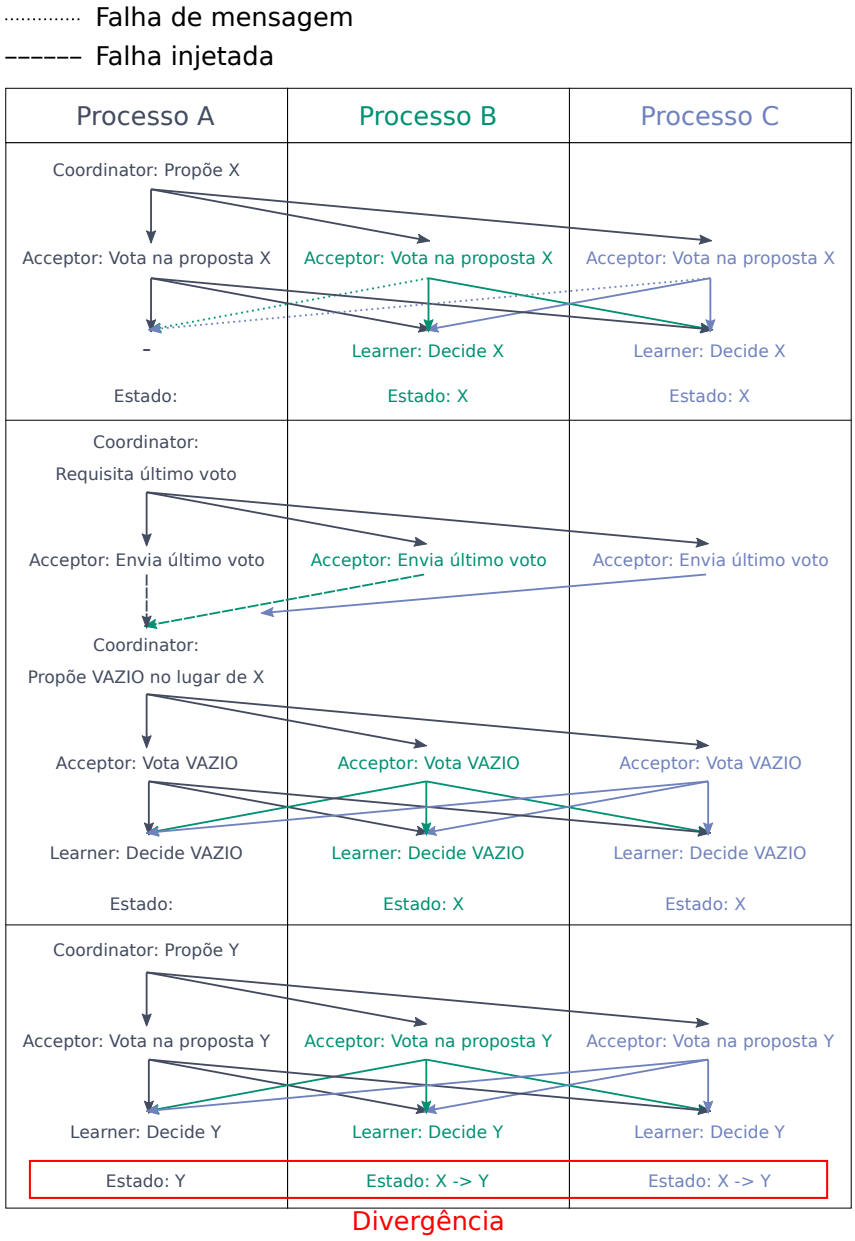
nada foi decidido e o mesmo cumpre papel de coordenador, inicia uma nova rodada para tentar alcançar uma decisão. O *Coordinator* retorna à fase de preparação e requisita aos *Acceptors* o valor votado na rodada em questão, e então a injeção da falha alvo faz com que os processo A e B esqueçam seus votos e enviem um valor nulo. Nesse momento, se os votos nulos dos processos A e B chegam antes do voto com a proposta X do processo C, será formado um quórum de votos nulos e o coordenador começará a agir solicitando uma nova votação para decidir outra proposta no lugar de X, ignorando a mensagem do último *Acceptor* que pode ou não chegar. No caso da implementação do Treplica, na ausência de um valor válido para ser decidido é escolhida uma proposta vazia, ou seja, sem ações. Se a proposta vazia for decidida sem problemas por todos os processos, o algoritmo prosseguirá para a próxima instância decidindo uma nova proposta qualquer, por exemplo a proposta Y. A proposta Y sendo decidida e entregue a todas máquinas de estados haverá uma divergência entre os processos B e C e o processo A, já que os dois primeiros enviaram às suas máquinas de estado as ações contidas em X antes de Y, enquanto para A as ações da proposta X nunca ocorreram. Nesse caso, o processo A diverge do quórum tornando-se defeituoso.

3.2.3 Learner decide propostas sem um quórum

Foi desenvolvida uma falha que, quando injetada, faz com que o *Learner* decida uma proposta mesmo que ainda não tenha recebido um quórum de votos para ela. Esta falha, combinada com a falha de mensagem para impedir que rodadas de decisão terminem com sucesso, é capaz de gerar um erro semelhante ao detalhado na Figura 18, onde após uma sequência de falhas o Processo A diverge dos demais.

A injeção de falhas de mensagem causa um problema na comunicação entre os processos, fazendo com que uma proposta X enviada pelo coordenador na fase de aceitação seja recebida por apenas um dos *Acceptors*, fazendo com que os *Learners* recebam apenas um voto e nada seja decidido. Após um intervalo de tempo sem que nada ocorra, o *Coordinator* iniciará uma nova rodada e solicitará na fase de preparação que os processos digam em que propostas votaram na última rodada de consenso. Assim que é recebido um quórum de respostas o coordenador passará para a fase de aceitação, e caso esse quórum seja formado pelos dois processos que não receberam o pedido de voto na proposta X, a ação do coordenador será realizar uma nova votação com uma proposta vazia. Caso a proposta vazia seja decidida com sucesso o algoritmo prosseguirá, realizando novas rodadas de consenso com novas propostas, como por exemplo a proposta Y. Todo esse comportamento é esperado e tolerado pelo algoritmo Paxos, no entanto, suponha a falha alvo foi injetada no processo B, permitindo que o mesmo decida uma proposta com apenas um voto recebido, ou seja, sem a necessidade de um quórum. Nesse caso, a rodada que tentou decidir a proposta X e falhou terá decidido X no processo B, e à partir da próxima

Figura 17: Cenário de erro - *Acceptor* esquece seu último voto

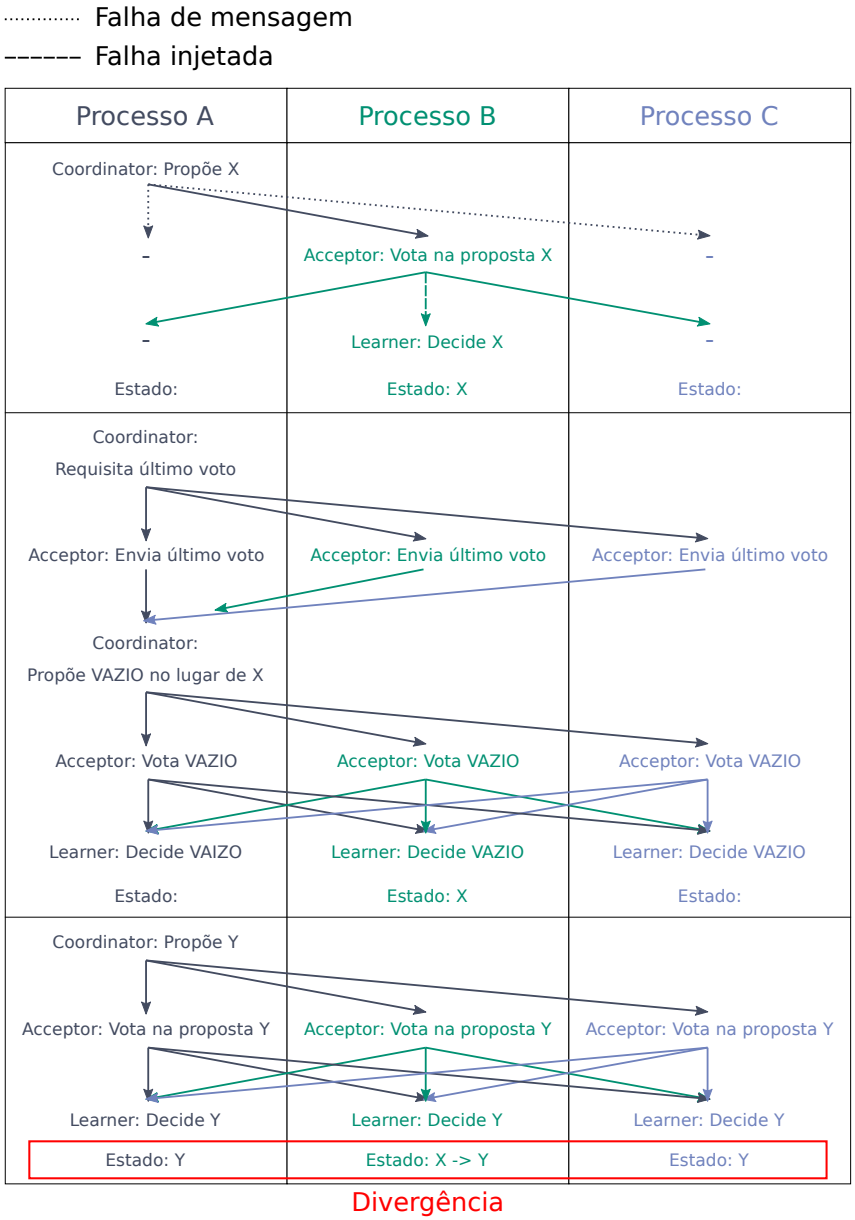


proposta decidida o estado desse processo estará divergente dos demais, caracterizando-se um processo defeituoso.

3.2.4 Coordinator esquece a proposta que deve ser decidida

Caso fique sem receber mensagens dos processos por um intervalo de tempo, o *Coordinator* inicia uma nova rodada de decisão do algoritmo Paxos com a mesma proposta da rodada anterior, a fim de tentar decidi-la novamente, assumindo que houve falha na comunicação durante a última tentativa. Para saber qual proposta estava sendo decidida, durante a fase de preparação, o coordenador requisita aos *Acceptors* seus últimos votos, e ao receber um quórum de votos para determinada proposta, tenta decidi-la novamente. Foi desenvolvida uma falha que, quando injetada, faz com que o coordenador esqueça a

Figura 18: Cenário de erro - *Learner* decide propostas sem um quórum



proposta para a qual recebeu um quórum de votos, propondo uma proposta vazia no lugar. Essa falha quando injetada junto com a falha de perda de mensagens, consegue simular um cenário de erro semelhante ao visto na Figura 19, onde o processo A ao final da última rodada ilustrada tem seu estado divergente por não possuir as ações da proposta X.

O coordenador tenta decidir uma proposta X em uma determinada rodada do algoritmo Paxos. Suponha que os *Acceptors* enviem os seus votos ao fim da fase de aceitação, no entanto devido a injeções da falha de mensagem os votos nunca chegam ao processo A, onde está localizado o coordenador. Desta forma, apenas os processos B e C decidem a proposta X enquanto o processo A não decide nada. O comportamento padrão do coordenador após um intervalo de tempo sem receber mensagens é de iniciar uma nova rodada e tentar decidir novamente a mesma proposta, a fim de que possíveis mensagens

que tenham se perdido sejam reenviadas e a proposta seja decidida com sucesso. Durante a fase de preparação da nova rodada o *Coordinator* solicitará o último voto dos *Acceptors*, e ao receber um quórum de votos iguais saberá o que deve ser decidido. No entanto, se após receber um quórum de votos a falha alvo for injetada, o coordenador esquecerá qual proposta deveria ser decidida, e solicitará a decisão de uma proposta vazia no lugar da proposta X. Se essa nova rodada for finalizada com sucesso, o algoritmo prosseguirá e novas propostas serão decididas nas rodadas seguintes, como a proposta Y. A partir do momento que a proposta Y for decidida e suas ações entregues às máquinas de estados de A, B e C, o processo A, que nunca realizou a entrega da proposta X à sua máquina de estados, se tornará divergente do quórum de réplicas, sendo considerado um processo defeituoso.

3.3 Gerador de carga

O gerador de carga descrito por Barbieri (2016) é uma aplicação responsável por interagir com o Treplica, fazendo o papel do cliente e realizando uma série de operações no sistema. Para este trabalho, esse gerador de carga original foi modificado principalmente no que diz respeito à forma como registra seus resultados. Com as alterações realizadas, cada processo de cada rodada do experimento possui um arquivo independente para registrar seus resultados finais. Por exemplo, o processo 3 da rodada de execução 12, ao final da execução, registra o estado final do ambiente em um arquivo identificado da seguinte forma:

```
print_3_out_12.out
```

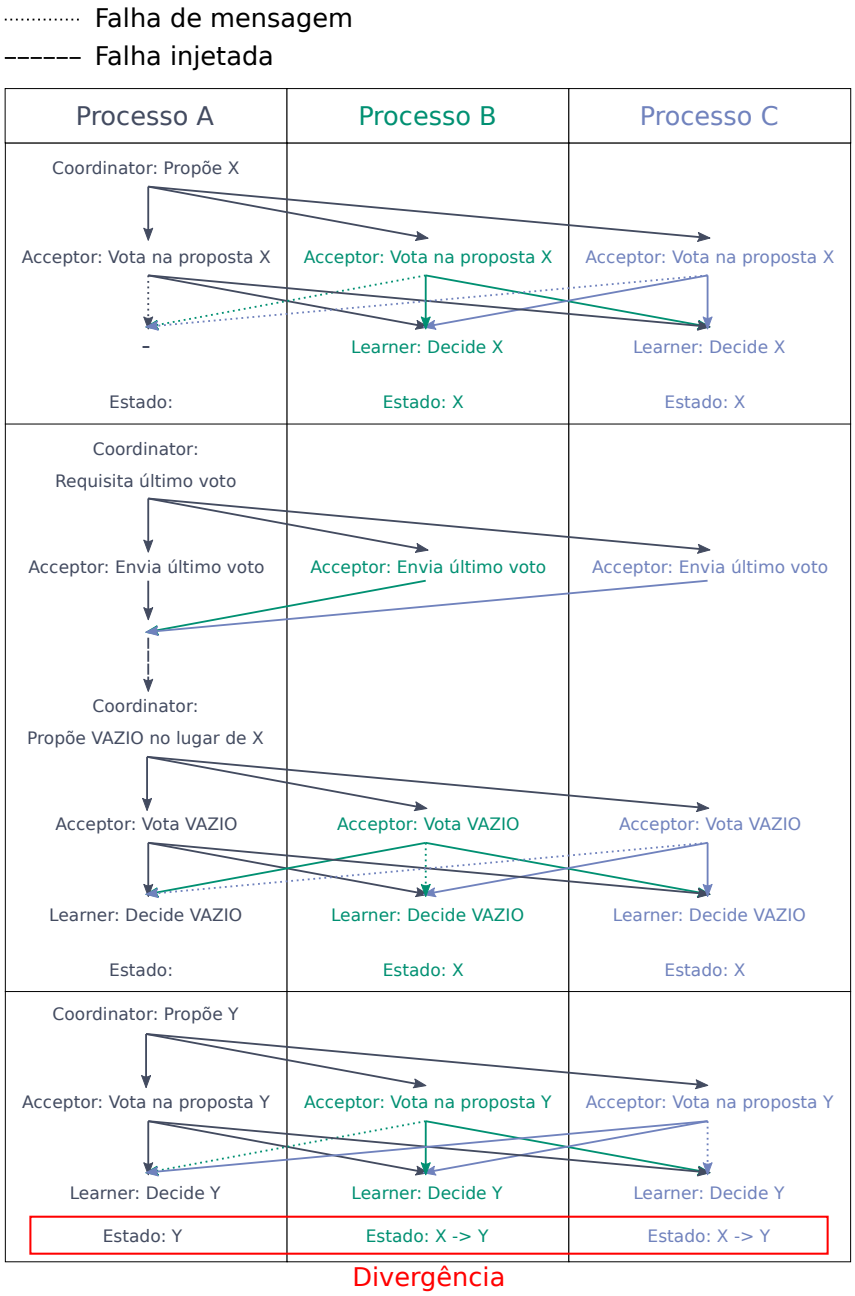
Esse registro é fundamental para que possam ser identificadas falhas que não foram toleradas, tendo seus erros propagados e por consequência tornado o serviço defeituoso. O modo de obter tal informação é comparando o estado final de cada processo em uma mesma rodada do experimento.

Além disso um segundo arquivo é gerado para cada execução em cada réplica, registrando a cada linha o tempo de início e fim de cada operação. Esse arquivo tem uma nomenclatura semelhante à do arquivo de resultado final, como é possível ver no exemplo abaixo, onde o processo 1 gerará um arquivo com o seguinte nome na execução 20:

```
20_out_1.log
```

Os registros desse arquivo permitem analisar o tempo que o sistema leva para realizar cada operação individualmente, e dessa forma medir o desempenho do algoritmo, realizar comparações e entender o impacto das modificações aplicadas.

Figura 19: Cenário de erro - *Coordinator* esquece a proposta que deve ser decidida



3.4 Parser dos Resultados

Todas execuções geram uma grande quantidade de registros durante sua execução em um arquivo de *log*, criando um histórico de tudo que ocorre durante a execução do algoritmo. Para evitar a oneração dos recursos, os eventos que não são interessantes para o teste, como as trocas de mensagem do algoritmo Paxos, foram ocultadas. São mantidos apenas os eventos de injeção de falha, de encerramento forçado do processo causado pelas validações e de fim de teste, quando a execução termina sem nenhum problema.

É necessário também analisar os arquivos gerados em cada réplica ao fim da execução do gerador de carga, onde são registrados todos os dados existentes na aplicação cliente no estado final da execução do teste. Esse aquivo, dependendo do número de operações por segundo e tempo de execução parametrizados, pode se tornar muito extenso.

Levando em consideração o volume de dados a serem analisados e a precisão necessária nos resultados tornou-se inviável fazer uma análise manual dos arquivos, o que levou à busca da automatização do processo de extração das informações necessárias para gerar métricas confiáveis dos resultados. Para isso, um *script* utilizando a linguagem Python[2] foi desenvolvido, tornando possível interpretar os arquivos e responder as seguintes perguntas em cada execução:

3.4.1 Em quantas execuções a validação desenvolvida foi responsável por tolerar as falhas?

Como já dito, o algoritmo Paxos por si só já é capaz de tolerar falhas no modelo *fail-recovery*, o que significa que certas falhas serão toleradas naturalmente, e não através da validação distribuída. É importante sabermos distinguir tais episódios daqueles quando a validação criada foi responsável pela tolerância, pois apenas esses últimos nos interessam para medir a efetividade da validação distribuída.

Para fazer essa distinção um evento especial foi registrado no arquivo de *log* quando um processo divergente tem sua execução interrompida pela validação distribuída, identificado através da seguinte frase: "*State diverged*". Assim, toda vez que o *parser* encontrar tal frase nos registros ele sabe que aquele processo foi interrompido por possuir um estado corrompido segundo os critérios da validação distribuída.

3.4.2 Em quantas execuções alguma falha não foi tolerada, gerando um estado defeituoso no serviço?

Como existem falhas que serão toleradas pelo algoritmo Paxos sem uso da validação distribuída, é natural que hajam execuções onde não há a manifestação da validação, pois

[2] Documentação disponível em: <https://bit.ly/3eqHFaF>

ela nunca precisou agir. No entanto, não se pode supor que em todas execuções onde a validação não interrompeu o processo as falhas foram toleradas pelo algoritmo. É necessário considerar a situação em que a falha tornou o sistema defeituoso, propagando seu erro apesar das validações. Para distinguir estas situações é necessário uma forma de validar ao final da execução se ela foi corrompida por algum erro ou não.

Isso é possível através dos arquivos gerados em cada réplica através do gerador de carga, que registra os dados finais de cada processo ao final de sua execução. Todos processos que chegaram até o final do teste sem ser interrompido, caso não estejam corrompidos, devem possuir o mesmo conteúdo em tal arquivo, logo a comparação de tal conteúdo entre todas réplicas é suficiente para fazer essa distinção. Réplicas cujo resultado final não siga o mesmo de um quórum são consideradas corrompidas, logo houveram falhas que não foram toleradas.

3.5 Parâmetros dos experimentos

Para que fosse possível extrair métricas válidas dos experimentos foi realizada uma padronização dos parâmetros de teste, tanto no que diz respeito às configurações das máquinas onde eles foram executados quanto às porcentagens de falhas injetadas em cada teste.

3.5.1 Ambiente

Os experimentos foram realizados na infraestrutura de *cloud* da Universidade Federal de São Carlos[3], em máquinas virtuais com as seguintes especificações técnicas:

- 4GB de memória RAM
- 2 CPUs
- 12GB de Disco
- CentOS 7 x86_64
- Java Development Kit (JDK) e Java Runtime Environment (JRE) versão 1.8.0 *update* 121

3.5.2 Cenários de teste

Ao todo 5 cenários de teste foram criados com os três tipos de injeção de falha descritos na Seção 3.2, detalhados na Tabela 1. Os testes 1 e 3 e os testes 2 e 4 usam as

[3] Disponível em: <https://bit.ly/2VBRR7B>

Tabela 1: Cenários de teste

Teste	Falha injetada	Réplicas com injeção de falha
1	*Acceptor* esquece seu último voto (3.2.2)	1 (20%)
2	*Learner* decide propostas sem um quórum (3.2.3)	1 (20%)
3	*Acceptor* esquece seu último voto (3.2.2)	5 (100%)
4	*Learner* decide propostas sem um quórum (3.2.3)	5 (100%)
5	*Coordinator* esquece a proposta que deve ser decidida (3.2.4)	5 (100%)

mesmas falhas, tendo como diferença que nos testes 1 e 2 apenas uma réplica possui falhas injetadas, enquanto nos testes 3 e 4 todas réplicas sofrem falhas. É importante notar que a falha do teste 5 só foi testada no cenário em que são injetadas falhas em todas réplicas, já que apenas uma delas será coordenadora por vez, logo apenas uma sofrerá falhas. Se apenas uma réplica for configurada para ter injeções, em um ambiente com 5 réplicas, aproximadamente 80% das execuções terminariam sem que nenhuma falha fosse injetada, já que algum dos outros 4 processos seria escolhido como coordenador nessas execuções. Além das falhas alvos, para todos os testes todas réplicas tiveram falhas de mensagem injetadas.

3.5.3 Probabilidades de injeções de falhas

O *framework* de injeção apresentado na Seção 3.1 necessita que seja fornecida uma probabilidade para que cada falha que deseje ser injetada ocorra. Por exemplo, para injetar a falha descrita na Seção 3.2.2 é necessário especificar para cada vez que a ação geradora da falha for executada a probabilidade dessa ocorrer, onde 0% significa que a falha nunca ocorrerá e 100% significa que a falha ocorrerá em todas as chamadas do método.

Através das descrições fornecidas na Seção 3.2 é possível notar como as falhas injetadas por si só não são capazes de gerar erros, dependendo de uma combinação destas com as falhas de perda de mensagem para desviar o Paxos do seu regime. Por necessitar de combinações de falhas menos complexas, algumas dessas falhas acabam gerando erros com mais facilidade do que outras, se considerarmos a mesma probabilidade de injeção para todas. Por esse motivo, torna-se difícil padronizar as métricas de injeção, e a solução encontrada foi variar as probabilidades de injeção de falha para cada experimento a fim de padronizar as probabilidades de ocorrências de erro devido à falha injetada.

Foi decidido calibrar cada probabilidade para que, em cada teste, em 20% das execuções não houvessem falhas que causassem divergência de estado em uma réplica, ou seja, que necessitassem da validação distribuída. Cada bateria de teste foi composta de 50 execuções, o que significa que por volta de 10 delas não deveriam possuir tais erros, enquanto aproximadamente 40 possuiriam erro em uma ou mais réplicas. Isto resulta em uma probabilidade de aproximadamente 3,17% de uma execução possuir um ou mais erros. É possível entender tais probabilidades através da equação 3.1, onde k é a porcentagem que desejamos atingir de execuções sem erros, que no caso foi decidido por 20%, e p a

Tabela 2: Probabilidades de injeções de falhas

Teste	**Falha injetada**	**Probabilidade de injeção de falhas**
1	*Acceptor* esquece seu último voto (3.2.2) - uma réplica	99%
	Falha no *payload* da mensagem (3.2.1) - todas réplicas	20%
2	*Learner* decide propostas sem um quórum (3.2.3) - uma réplica	80%
	Falha no *payload* da mensagem (3.2.1) - todas réplicas	20%
3	*Acceptor* esquece seu último voto (3.2.2) - todas réplicas	90%
	Falha no *payload* da mensagem (3.2.1) - todas réplicas	10%
4	*Learner* decide propostas sem um quórum (3.2.3) - todas réplicas	80%
	Falha no *payload* da mensagem (3.2.1) - todas réplicas	10%
5	*Coordinator* esquece a proposta que deve ser decidida (3.2.4) - todas réplicas	30%
	Falha no *payload* da mensagem (3.2.1) - todas réplicas	1%

probabilidade de uma execução possuir erros.

$$\begin{aligned} (1-p)^{50} &= k \\ 1-p &= \sqrt[50]{k} \\ p &= 1-\sqrt[50]{k} \\ p &= 1-\sqrt[50]{0.2} \\ p &= 1-0.9683 \\ p &= 0.0317 \end{aligned} \tag{3.1}$$

Levando em consideração todos esses aspectos, foi feito um levantamento experimental dos melhores parâmetros de probabilidade de injeção de falhas para cada teste. Os resultados obtidos foram detalhados na Tabela 2.

3.6 Testes de desempenho

Além de medir a efetividade das modificações propostas no algoritmo é interessante saber qual é o impacto de tais alterações no que diz respeito ao desempenho, ou seja, no número de operações que são realizadas em relação ao tempo de execução. Para isso, uma bateria de testes de 50 execuções sem falhas injetadas foi realizada, e os arquivos gerados pelo gerador de carga que registram o início e final de cada operação foi utilizado para extrair as métricas.

Para entender o impacto das mudanças é necessário ter uma base de comparação, sendo assim a bateria de testes também foi executada no Treplica original apresentado por Vieira e Buzato (2008) e na versão proposta por Barbieri (2016) utilizando os mesmos parâmetros tanto para o Treplica quanto para o gerador de carga.

3.7 Resultados

Após a execução dos testes de falha e desempenho e análise dos resultados foi possível chegar a conclusões quanto à efetividade e viabilidade de uso do algoritmo. Abaixo

Tabela 3: Resultados dos testes de falha

Teste	**Falha injetada**	**Falhas toleradas**			**Erros**
		0 processos	1 processo	2 processos	
1	*Acceptor* esquece seu último voto (3.2.2) - uma réplica	19	16	15	0
2	*Learner* decide propostas sem um quórum (3.2.3) - uma réplica	8	21	21	0
3	*Acceptor* esquece seu último voto (3.2.2) - todas réplicas	12	7	31	0
4	*Learner* decide propostas sem um quórum (3.2.3) - todas réplica	9	7	34	0
5	*Coordinator* esquece a proposta que deve ser decidida (3.2.4) - todas réplicas	13	5	32	0

serão expostos os resultados finais sumarizados para cada bateria de testes assim como uma breve análise do seu significado.

3.7.1 Testes de tolerância a falhas

Para cada teste foi levantado quantos processos possuíram falhas arbitrárias não maliciosas que necessitaram da validação proposta para serem toleradas, e ao final da execução quantos processos possuíam erros, ou seja, seus resultados finais divergiam da maioria. Este último caso representa processos que tiveram divergência de estado que não foi detectada pela proposta de validação. A Tabela 3 mostra os números contabilizados para as 50 execuções de cada cenário de teste.

Como já detalhado na Seção 3.5.3, os testes foram parametrizados para que o número de execuções com 0 processos apresentando falhas que necessitam da intervenção da validação distribuída fosse próximo de 20%. Como é possível notar, o Teste 1 foi o que ficou mais distante deste valor devido às características da falha testada, que a torna difícil de ser estimulada injetando falhas em apenas uma réplica.

Duas observações são importantes sobre os resultados obtidos para validar a eficácia do algoritmo proposto. A primeira delas é que nenhum dos testes gerou erros, ou seja, todos processos que executaram até o final sem que fossem interrompidos possuíam o mesmo estado, tendo entregue apenas valores validados pelo quórum à aplicação cliente. Isso, em conjunto com o fato de que a validação precisou intervir em aproximadamente 80% das execuções para prevenir corrupções no sistema, mostra que a validação de fato conseguiu tolerar as falhas propostas.

A segunda observação importante a ser feita é que não houveram para nenhuma bateria de testes casos em que mais de 2 processos apresentaram falhas que necessitassem da validação distribuída. Esta ocorrência seria problemática pois em um ambiente com 5 réplicas onde 3 foram interrompidas por causa da validação não é mais possível formar um quórum, logo o processamento deixa de progredir. Isso se justifica pelas características das falhas injetadas e do algoritmo Paxos que necessitam de réplicas suficientes para gerar dois quóruns, um para a decisão correta e outro para a decisão ocasionada pela falha. A intersecção desses dois quóruns precisa possuir ao menos um processo diferente, onde o processo que está no quórum da falha e não esteve no quórum correto terá seu estado divergente. Com réplicas suficientes apenas para um quórum todas ações dependem da

Tabela 4: Resultados dos testes de desempenho

Versão	**Op/s**	**Desvio padrão (op/s)**	**Latência (ms)**	**Desvio padrão (latência/ms)**
Paxos original (VIEIRA; BUZATO, 2008)	1299	191	615	495
Paxos arbitrário não malicioso (BARBIERI, 2016)	687	90	1245	910
Paxos com a nova validação distribuída	657	107	1546	1713

ciência de todas réplicas, o que diminui a possibilidade de erros acontecerem, ao mesmo tempo que diminui a tolerância a falhas do sistema (um colapso e o mesmo travará).

3.7.2 Testes de desempenho

Além de entender a eficácia do algoritmo perante a falhas, é interessante saber o impacto das modificações no que diz respeito ao desempenho pois o intuito é que o algoritmo seja viável para uso. O desempenho foi analisado medindo o tempo completo para realizar uma operação, ou seja, a diferença de tempo entre o momento que a aplicação solicita uma ação e o resultado da mesma é entregue à aplicação, já decidida e efetivada na máquina de estados.

A fim de entender o impacto das modificações é necessário ter uma base de comparação, e para isso foram utilizadas as versões do algoritmo Treplica propostas por Vieira e Buzato (2008) e Barbieri (2016), a primeira delas contendo o algoritmo Paxos sem modificações e a segunda com as validações citadas na Seção 1.4. Os resultados das baterias de 50 execuções para cada versão foram resumidos na Tabela 4.

É possível notar um impacto considerável na aplicação, com uma queda de quase 50% na performance, quando analisado em comparação ao algoritmo Paxos original apresentado por Vieira e Buzato (2008), onde as falhas são toleradas no modelo *fail-recovery*. Esta diferença é estatisticamente significativa, usando-se um teste-t independente, com $p < 0.001$ ($t_{(50)} = 20.736$). No entanto, ao analisarmos as duas implementações com as mesmas suposições de falha, ou seja, os dois algoritmos que propõe tolerância a falhas no modelo arbitrário não malicioso, notamos uma diferença sutil entre o algoritmo produzido por Barbieri (2016) e o algoritmo proposto. Com essa diferença, de cerca de 4%, é possível concluir que a nova validação distribuída desenvolvida durante esta dissertação tem um impacto sutilmente maior do que a descrita na Seção 1.4.4. Esta pequena diferença não é estatisticamente significativa, usando-se um teste-t independente, com $p = 0.1332$ ($t_{(50)} = 1.517$). É importante salientar que tais valores podem variar de acordo com a aplicação cliente utilizada e o intuito de uso do Treplica.

Conclusão

Após o estudo da bibliografia destacou-se a versão do algoritmo Paxos apresentada em Barbieri (2016) que propõe uma tolerância a falhas no modelo arbitrário não malicioso. No entanto, observando os resultados do trabalho, o algoritmo não é totalmente tolerante às categorias de falhas propostas, o que se confirmou ao replicar os experimentos. Após uma análise do comportamento dos experimentos foi possível entender a causa raiz das corrupções de estado que ocorreram.

O principal problema se encontrava na validação distribuída exposta na Seção 1.4.4, mais precisamente no fato de ela aguardar uma confirmação da corrupção de estado para interromper um processo defeituoso. Este comportamento permite que o mesmo envie mensagens para as demais réplicas, o que vai contra a propriedade de honra do modelo computacional, como explicado na Seção 1.1.2.2. A fim de deixar o algoritmo em conformidade com o modelo proposto, a dissertação aqui exposta propôs uma nova versão da validação distribuída, onde a máquina de estados entra em um estado de espera antes de consolidar um novo estado até que ele seja validado perante um quórum de réplicas. Dessa forma evita-se que estados corrompidos sejam enviados à aplicação e que o processo defeituoso influencie em rodadas de consenso.

Para que a implementação prática da validação fosse possível diversas modificações precisaram ser feitas nos componentes do Paxos, assim como na máquina de estados. Uma nova camada que interage com essas duas foi criada, chamada de *Authentication*, responsável por centralizar todas as operações da validação distribuída e servindo como ponte entre o Paxos e a máquina de estados, já que a interação dessas duas camadas torna-se necessária.

Experimentos foram executados tanto para medir a efetividade da solução quanto seu desempenho em relação a outras versões do Paxos. Quanto à efetividade, categorias de falha foram criadas nos componentes vitais do Paxos e foi possível notar que em todos os cenários as falhas foram toleradas, não propagando nenhum erro para a aplicação até o final de cada execução, comprovando a tolerância a falhas no modelo arbitrário não malicioso. Já quanto ao desempenho, foi notada uma queda de quase 50% quando comparado com o algoritmo proposto por Vieira e Buzato (2008), que implementa o Paxos original, quando levado levado em consideração o número de operações concretizadas por segundo. No entanto, o algoritmo Paxos original é tolerante a falhas apenas no modelo *fail-recovery* descrito na Seção 1.1.1.1. Quando comparado com o algoritmo de Barbieri (2016) que se propõe a tolerar falhas no mesmo modelo da dissertação, porém com algumas deficiências, a queda de performance é sutil, não sendo estatisticamente significativa.

Com esse trabalho foi possível obter uma versão do algoritmo Paxos tolerante a falhas no modelo arbitrário não malicioso, apresentando 100% de efetividade para todas as falhas testadas. O processo de tolerância envolve a interrupção do processo defeituoso para que o mesmo não polua o ambiente. Sendo assim, dado tempo suficiente para as falhas ocorrerem, em dado momento o *cluster* terá a quantidade mínima de réplicas necessárias para formar um quórum, o que pela forma como o Paxos funciona diminuirá naturalmente as chances de uma nova falha causar um erro, mas não extingue essa possibilidade. Logo, no próximo erro que cause a interrupção de um processo o sistema deixará de ter processos suficientes para formar um quórum e deixará de realizar operações, pois não é possível decidir novas propostas. Levando isso em consideração uma proposta de trabalho futuro que complementaria o algoritmo desenvolvido seria o desenvolvimento de um mecanismo para correção automática, quando possível, dos processos defeituosos. Assim, os processos poderiam ser interrompidos, corrigidos e reintegrados ao Paxos, diminuindo consideravelmente a chance do sistema tornar-se indisponível por falta de réplicas suficientes.

Publicações

BARBIERI, R.; SANTOS, E. dos; VIEIRA, G. M. D. Decentralized validation for non-malicious arbitrary fault tolerance in paxos. In: *Anais do XX Workshop de Testese Tolerância a Falhas*. Porto Alegre, RS, Brasil: SBC, 2019. p. 34–47. ISSN 2595-2684. Disponível em: <https://sol.sbc.org.br/index.php/wtf/article/view/7713>.

Referências

AVIZIENIS, A. et al. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, IEEE, v. 1, n. 1, p. 11–33, 2004. Citado na página 24.

BARBIERI, R. R. *Achieving non-malicious arbitrary fault tolerance in Paxos through hardening techniques*. Dissertação (Mestrado) — Universidade Federal de São Carlos, 2016. Citado 14 vezes nas páginas 25, 33, 34, 35, 36, 37, 39, 43, 44, 48, 54, 59, 61 e 63.

BARBIERI, R. R.; VIEIRA, G. M. D. Hardened paxos through consistency validation. In: *SBESC '15: Proceedings of the V Brazilian Symposium on Computing Systems Engineering*. Foz do Iguaçu, Brazil: IEEE Computer Society, 2015. (SBESC '15), p. 13–18. Citado 4 vezes nas páginas 34, 35, 36 e 37.

BEHRENS, D.; WEIGERT, S.; FETZER, C. Automatically tolerating arbitrary faults in non-malicious settings. In: IEEE. *Dependable Computing (LADC), 2013 Sixth Latin-American Symposium on*. [S.l.], 2013. p. 114–123. Citado 3 vezes nas páginas 24, 25 e 28.

BHATOTIA, P. et al. Reliable data-center scale computations. In: ACM. *Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware*. [S.l.], 2010. p. 1–6. Citado na página 34.

CACHIN, C.; GUERRAOUI, R.; RODRIGUES, L. *Introduction to reliable and secure distributed programming*. [S.l.]: Springer Science & Business Media, 2011. Citado 2 vezes nas páginas 24 e 27.

CASTRO, M.; LISKOV, B. et al. Practical byzantine fault tolerance. In: *OSDI*. [S.l.: s.n.], 1999. v. 99, p. 173–186. Citado na página 24.

CORREIA, M. et al. Practical hardening of crash-tolerant systems. In: *USENIX Annual Technical Conference*. [S.l.: s.n.], 2012. v. 12. Citado na página 24.

LAMPORT, L. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, ACM, v. 16, n. 2, p. 133–169, 1998. Citado 2 vezes nas páginas 23 e 29.

LAMPORT, L. Fast paxos. *Distributed Computing*, Springer, v. 19, n. 2, p. 79–103, 2006. Citado na página 30.

SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, ACM, v. 22, n. 4, p. 299–319, 1990. Citado 4 vezes nas páginas 23, 24, 29 e 30.

VIEIRA, G. M.; BUZATO, L. E. Treplica: ubiquitous replication. In: CITESEER. *SBRC'08: Proc. of the 26th Brazilian Symposium on Computer Networks and Distributed Systems*. [S.l.], 2008. p. 25. Citado 5 vezes nas páginas 30, 46, 59, 61 e 63.

VIEIRA, G. M.; BUZATO, L. E. Implementation of an object-oriented specification for active replication using consensus. *Technical Report IC-10–26*, 2010. Citado na página 30.