Rodrigo Herpich Müller

Arcabouço para desenvolvimento de aplicações com Replicação Máquina de Estados

Rio Grande

2017

Rodrigo Herpich Müller

Arcabouço para desenvolvimento de aplicações com Replicação Máquina de Estados

Monografia submetida à Universidade Federal do Rio Grande - FURG, como requisito parcial à obtenção do grau de Bacharel em Sistemas de Informação.

Universidade Federal do Rio Grande - FURG
Centro de Ciências Computacionais
Sistemas de Informação

Orientador: Odorico Machado Mendizabal

Rio Grande
2017

Rodrigo Herpich Müller

Arcabouço para desenvolvimento de aplicações com Replicação Máquina de Estados

Monografia submetida à Universidade Federal do Rio Grande - FURG, como requisito parcial à obtenção do grau de Bacharel em Sistemas de Informação.

Rio Grande, 29 de novembro de 2017

Orientador
Dr. Odorico Machado Mendizabal

Professor
Dr. Nelson Lopes Duarte Filho

Professora
Dra. Diana Francisca Adamatti

Rio Grande
2017

Este trabalho é dedicado aos meus pais e à minha irmã, por me ajudarem a chegar até aqui.

Agradecimentos

Primeiramente, um agradecimento muito especial aos meus pais, Ademir e Ivete, pelo suporte que me permitiu realizar uma graduação longe de casa, com tudo que isso acarreta. À minha irmã, Ingrid, pelo apoio durante o curso, principalmente na elaboração deste trabalho.

Ao meu orientador, prof. Odorico, pela ajuda quando eu tive dificuldade em entender algum tópico deste trabalho, que muitas vezes se mostraram desafiadores, e pelas revisões dos textos, que permitiram elevar a qualidade desta monografia.

A todos que, direta ou indiretamente, me ajudaram nesta monografia, ou durante a graduação.

Nem todos os que vagueiam estão perdidos
(J. R. R. Tolkien)

Resumo

Replicação Máquina de Estados (*State Machine Replication* – SMR) é uma estratégia para o desenvolvimento de sistemas com alta disponibilidade, na qual as réplicas do serviço recebem e executam todas as requisições vindas dos clientes de forma ordenada. Um dos aspectos centrais é que deve haver consenso na ordem de execução de comandos entre as réplicas, para evitar inconsistências. Porém, ao desenvolver aplicações usando SMR, normalmente é difícil desassociar a lógica da aplicação do protocolo de comunicação. O objetivo deste trabalho é desenvolver um arcabouço para SMR que abstraia os detalhes de implementação relacionados a garantias de ordem e entrega de mensagens. Desta forma, desenvolvedores podem se dedicar à implementação das aplicações sem envolvimento com aspectos relacionados à tolerância a falhas.

Palavras-chave: Replicação Máquina de Estados, Desenvolvimento de Software, Tolerância a falhas.

Abstract

State Machine Replication is a strategy to develop systems with high availability, in witch the service replicas receive and execute all the requests from clients in a orderly manner. One of the keys aspects is that a consensus is required in the execution order of commands between the replicas, to avoid inconsistencies. However, when developing applications using SMR, usually it's hard to disassociate the business logic from the communication protocol. This project's goal is develop a SMR framework that abstract the implementation details related to guarantees of order and messages delivery. Thus, developers can dedicate to implementation of applications without involvement with aspects related to fault tolerance.

Keywords: *State Machine Replication, Software Development, Fault Tolerance*

Lista de ilustrações

Figura 1 – Distinção entre falha, defeito e erro. Adaptado de (WEBER, 2003) . . 16
Figura 2 – Hierarquia de falhas. Fonte: (GÄRTNER, 1998) 17
Figura 3 – Representação Replicação Passiva. Fonte: autor 19
Figura 4 – Divisão das fases no algoritmo Paxos. Fonte: autor 23
Figura 5 – Modelo conceitual do arcabouço 32
Figura 6 – Diagrama UML da Classe Setup 33
Figura 7 – Diagrama UML da ClientInterface 35
Figura 8 – Arquivo modulosCliente.properties 36
Figura 9 – Gráfico de desempenho do S-Paxos com o arcabouço 45

Lista de tabelas

Tabela 1 – Comparativo entre as implementações 29
Tabela 2 – Comparativo Reflexão 44
Tabela 3 – Comparativo de desempenho na biblioteca S-Paxos 45
Tabela 4 – Diferença percentual utilizando-se o arcabouço 45

Lista de abreviaturas e siglas

SMR	*State Machine Replication*
VR	*Viewstamped Replication*
RPC	*Remote Procedure Call*
IP	*Internet Protocol*

Sumário

Introdução **13**

1 FUNDAMENTAÇÃO TEÓRICA **15**
1.1 Definição de falhas **15**
1.1.1 Modelos de falhas 16
1.2 Replicação **17**
1.2.1 Replicação Passiva 18
1.2.2 Replicação Ativa 19
1.2.3 Abordagens híbridas 20
1.3 Protocolos de Ordenação **20**
1.3.1 Difusão atômica 21
1.3.2 Paxos 21
1.3.3 Raft 23
1.3.4 Viewstamped Replication 25
1.4 Técnicas de Programação **26**
1.4.1 Padrões de Projeto 27
1.4.2 Reflexão 27
1.4.3 Anotações 28

2 IMPLEMENTAÇÕES DE PAXOS **29**
2.1 BFT-SMaRt **29**
2.2 S-Paxos **30**
2.3 LibPaxos **30**
2.4 OpenReplica **31**

3 ARCABOUÇO PARA REPLICAÇÃO MÁQUINA DE ESTADOS . . **32**
3.1 Criando novas réplicas **33**
3.2 Criando novos clientes **35**
3.2.1 Pacote Services 37
3.2.2 Anotação Replicar 37
3.3 Módulos **37**
3.3.1 Réplica Genérica 38
3.3.2 Cliente Genérico 39
3.4 Resumo **40**

4 RESULTADOS **41**

5 CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS 47

REFERÊNCIAS 48

Introdução

A necessidade de sistemas tolerarem falhas é uma constante, principalmente pela crescente importância de sistemas computacionais. Assim, a disponibilidade destes passa a ser não um diferencial competitivo, mas um requisito importante, já que a indisponibilidade destes serviços pode acarretar em perda de clientes e, por conseguinte, de faturamento.

Um dos meios de prover tolerância a falhas é através da replicação do serviço. Assim, se uma das instâncias do serviço em execução torna-se inoperante, as demais ainda continuam atendendo as requisições. Há diversas abordagens que podem ser utilizadas para implementar esta replicação, sendo as mais comuns a abordagem passiva, ou *primary-backup*, e a ativa, também conhecida como *State Machine Replication* (SMR), ou Replicação Máquina de Estados (WIESMANN et al., 2000). Nesta abordagem, todas as réplicas recebem e executam todas as requisições dos clientes, na mesma ordem (SCHNEIDER, 1990). A garantia de ordem na entrega das mensagens pode ser obtida por protocolos de difusão atômica (DÉFAGO; SCHIPER; URBáN, 2004), ou consenso. Segundo (LAMPORT, 2001), um protocolo de consenso garante que apenas um valor, dentre os propostos, é escolhido. Um dos algoritmos de consenso mais utilizados na construção de SMR é o Paxos (LAMPORT, 1998), sendo utilizado, entre outros, no Google e na Microsoft (CHANDRA; GRIESEMER; REDSTONE, 2007a; RAO; SHEKITA; TATA, 2011).

O artigo original do protocolo, proposto por Lamport em (LAMPORT, 1998), foi considerado bastante desafiador, sendo apresentado de forma mais simples em (LAMPORT, 2001). Mesmo com as simplificações feitas neste artigo, este limita-se a explicar o problema do consenso, e uma possível solução. Detalhes de implementações são deixados, deliberadamente, fora do escopo do artigo. Conforme (CHANDRA; GRIESEMER; REDSTONE, 2007b), a transformação do algoritmo em um sistema prático, pronto para uso, envolveu a implementação de muitos atributos e otimizações, algumas presentes na literatura, outras não.

Esta dificuldade reflete-se, também, nas implementações de código-aberto do Paxos. O correto uso destas requer um entendimento de como determinada funcionalidade foi feita e, muitas vezes, entender a biblioteca utilizada para fazer a comunicação entre as réplicas. Este é o caso, por exemplo, do LibPaxos (LIBPAXOS, 2016), uma implementação desenvolvida na linguagem C, que troca as mensagens através da biblioteca Libevent (LIBEVENT, 2012). Para utilizar o LibPaxos, é necessário ter um conhecimento de como as mensagens são enviadas pelo Libevent.

O objetivo geral deste trabalho é o desenvolvimento de um arcabouço que facilite a implementação de aplicações utilizando Replicação Máquina de Estados. Além disso, é

desejável que o desenvolvedor do serviço possa selecionar qual protocolo de comunicação será utilizado, sem precisar refatorar o código para adequação à biblioteca escolhida. Os objetivos específicos incluem:

- Compreender as diferenças entre os algoritmos de consenso Paxos, Raft e Viewstamped Replication.
- Analisar as implementações de código-aberto do Paxos, bem como desenvolver aplicações como estudo de caso para estas implementações.
- Explorar boas práticas de desenvolvimento de *software*, como o uso de Padrões de Projeto, visando o desenvolvimento de um arcabouço modular, de boa manutenabilidade e extensibilidade.

A organização do presente trabalho está descrita a seguir. No Capítulo 1 são apresentados conceitos fundamentais para o trabalho. No Capítulo 2 são discutidas algumas implementações de Paxos de código aberto existentes. No Capítulo 3 é discutida a construção do arcabouço, descrevendo quais técnicas foram utilizadas bem como a exemplo de uso. No Capítulo 4, são analisados os resultados obtidos. Por fim, no Capítulo 5, é apresentada uma discussão sobre o trabalho.

1 Fundamentação Teórica

Este capítulo tem como objetivo agrupar os conceitos teóricos abordados neste trabalho, bem como descrever algumas técnicas de programação que serão utilizadas no arcabouço.

1.1 Definição de falhas

Sistemas computacionais devem, idealmente, ser livres de erros. Enquanto em alguns sistemas, estes erros podem causar um mero desconforto ao usuário, em outros, as consequências podem ser catastróficas. Um exemplo disto foi o foguete Ariane 5, que explodiu aproximadamente 40 segundos após seu lançamento. Em (ARIANE, 1996), a causa da explosão foi apontada como a completa perda de direção e informações de altitude, devido a erros de especificações e desenvolvimento no *software* do sistema de referência inercial. Embora este tipo de falha decorra de ação humana, outras podem ser resultados de problemas físicos. Assim, é virtualmente impossível termos um sistema que esteja livre de qualquer falha, em seu ciclo de vida.

Em sistemas distribuídos, existe a noção de falhas parciais. Segundo (TANENBAUM; STEEN, 2008, pág 194), "uma falha parcial pode acontecer quando um componente em um sistema distribuído falha. Essa falha pode afetar a operação adequada de outros componentes e, ao mesmo tempo, deixar outros totalmente ilesos". Já em ambientes centralizados, a ocorrência de uma falha quase sempre é total, já que afeta todos os componentes do sistema, fazendo com que este fique, usualmente, indisponível (TANENBAUM; STEEN, 2008, pág 194). Embora coloquialmente as palavras defeito, erro e falhas sejam utilizadas como sinônimos, cada uma delas têm um significado próprio. Defeito (*failure*) é, segundo (WEBER, 2003), um desvio da especificação. Já para (TANENBAUM; STEEN, 2008, pág 195), "um sistema apresenta defeito quando não pode cumprir suas promessas", isto é, se o sistema foi projetado para entregar a seus usuários determinado serviço, um defeito ocorre quando parte ou o todo do serviço não puder ser entregue. Ainda segundo Tanenbaum, erro "é uma parte do estado de um sistema que pode levar a uma falha". Por exemplo, no envio de pacotes por uma rede, é possível que alguns deles sejam perdidos. Por fim, a falha pode ser definida como a causa física ou algorítmica de um erro (WEBER, 2003).

É importante ressaltar que nem todo erro gera um defeito no sistema. No exemplo da perda de pacotes na comunicação pela rede, o defeito só existirá se as falhas que levam ao erro - a perda de pacotes, não for devidamente tratada. A Figura 1 sintetiza a distinção entre defeito, falha e erro. Nela, é possível observar que o defeito, perceptível ao usuário, decorre de um erro. Este erro, por sua vez, é originado por uma falha.

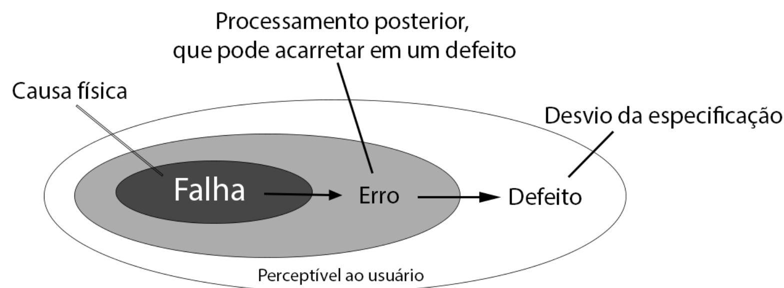

Figura 1 – Distinção entre falha, defeito e erro. Adaptado de (WEBER, 2003)

1.1.1 Modelos de falhas

Em um sistema distribuído, uma falha pode ocorrer por diversos motivos, tendo como exemplo problemas físicos na rede de transmissão, impedindo que determinadas mensagens sejam enviadas ou recebidas. Algumas destas falhas são mais facilmente detectáveis e tratáveis, enquanto outras requerem um esforço maior para que sejam toleradas. Neste trabalho, utilizou-se o modelo de falhas proposto em (GÄRTNER, 1998), como descrito a seguir.

- ***Failstop***: Neste modelo de falha, um processo falha por parada. Os demais processos, entretanto, conseguem detectar que tal processo parou de responder.
- ***Crash***: Também conhecido por falha por colapso, o processo falha, mas os demais processos do sistema não conseguem detectar a falha.
- ***Send Omission***: Na omissão de envio, o processo pode se comportar como em *crash*, ou pode falhar ao enviar apenas um subconjunto das mensagens que deveria.
- ***Receive Omission***: Este modelo de falha possui um comportamento bastante semelhante à falha por omissão de envio, porém, ao invés de somente enviar um subconjunto, na falha de recebimento, o processo pode falhar ao receber somente um subconjunto das mensagens destinadas a ele.
- ***General Omission***: Na falha por omissão, pode apresentar tanto a omissão de envio quanto a de recebimento.
- ***Byzantine***: O modelo de falhas bizantinas (LAMPORT; SHOSTAK; PEASE, 1982), ou arbitrárias, é o mais severo dos modelos. Nele, o processo pode gerar saídas defeituosas, ocasionadas por falhas benignas, ou maliciosas. Conforme (COULOURIS; DOLLIMORE; KINDBERG, 2007, pág. 196), “um servidor faltoso pode até estar trabalhando maliciosamente em conjunto com outros servidores para produzir respostas erradas intencionalmente”.

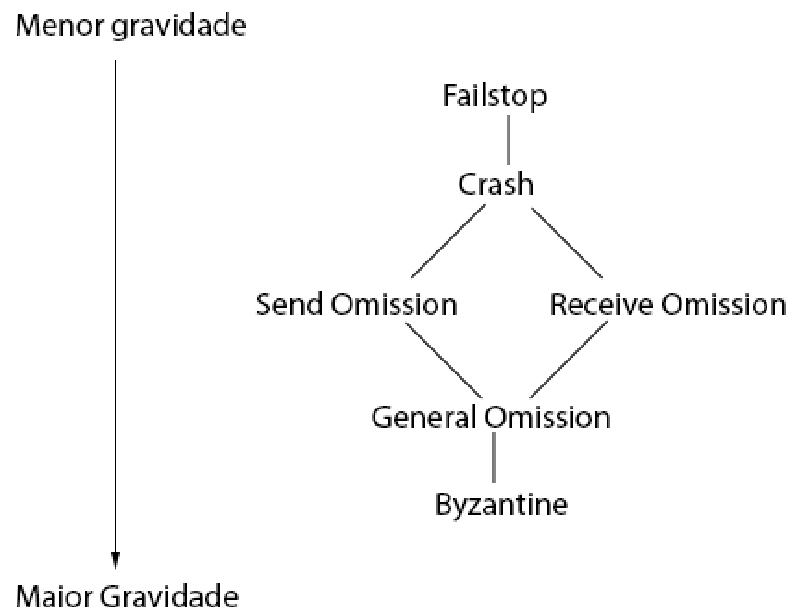

Figura 2 – Hierarquia de falhas. Fonte: (GÄRTNER, 1998)

Como evidenciado na Figura 2, cada camada da hierarquia engloba a anterior. Assim, falha por omissão pode comportar-se como *crash*. O que torna a falha bizantina muito mais complexa de tratar é que, ao contrário das anteriores, nenhuma asserção pode ser feita sobre a veracidade da resposta, se analisarmos um único componente. Isso ocorre porque, devido a erros de *software* e/ou *hardware*, ou até mesmo devido a invasões no sistema, a resposta pode estar incorreta, ingênua ou deliberadamente.

1.2 Replicação

A utilização de serviços replicados é motivada por pelo menos um destes três objetivos: melhorar o desempenho de um serviço, aumentar sua disponibilidade, ou torná-lo tolerante a falhas (COULOURIS; DOLLIMORE; KINDBERG, 2007, pág. 521). A melhoria de desempenho ocorre ao replicar os dados em diversas máquinas, distribuindo a carga entre elas. Quando estes dados são imutáveis, esta técnica é simples. Porém, ao lidar com dados dinâmicos, a troca de mensagens entre os servidores requeridas para garantir que o cliente receba as informações atualizadas é um limite na eficácia desta abordagem.

A maior disponibilidade do serviço decorre do fato de que, caso algumas réplicas ficarem inoperantes, outras ainda atendem as requisições dos usuários. Assim, dada uma probabilidade de uma réplica sofrer uma falha, e assumindo que as falhas entre os servidores são independentes, a disponibilidade de um sistema replicado pode ser dada por $1 - p^n$, sendo p a probabilidade de uma falha ocorrer em um servidor, e n a quantidade de servidores.

Por fim, há a motivação de tolerar falhas. Segundo (COULOURIS; DOLLIMORE; KINDBERG, 2007):

> “Dados de alta disponibilidade não são necessariamente dados rigorosamente corretos. Eles podem estar desatualizados, por exemplo, ou dois usuários em lados opostos de uma rede que foi particionada podem fazer atualizações conflitantes e que precisem ser resolvidas. Em contraste, um serviço tolerante a falhas sempre garante comportamento rigorosamente correto, apesar de certo número e tipos de falhas.”

Assumindo falhas do tipo *crash*, para garantir que o serviço permaneça tolerante a f falhas, deve haver $2f+1$ réplicas. Porém, se levarmos em consideração as falhas bizantinas, nas quais o componente passa a se comportar de forma arbitrária ou maliciosa, o número de réplicas necessárias é $3f+1$, para f falhas (COULOURIS; DOLLIMORE; KINDBERG, 2007).

Há diversas estratégias que podem ser utilizadas para prover replicação. A seguir as principais estratégias de replicação são apresentadas.

1.2.1 Replicação Passiva

Neste modelo, também conhecido como *primary-backup*, apenas uma réplica (primária) processa as requisições, e envia mensagens de atualizações às demais réplicas (*backups* ou escravas, em inglês *slave*) (DÉFAGO; SCHIPER, 2004). Caso a réplica primária falhe, alguma das réplicas secundárias assume este papel, passando a receber as requisições dos clientes.

Segundo (BUDHIRAJA et al., 1993), neste modelo de replicação, requisições podem ser perdidas, e que protocolos adicionais devem ser empregados para retransmitir estas requisições. Por outro lado, continua (BUDHIRAJA et al., 1993), a replicação passiva envolve menos processamento redundante, sendo menos custosa. Já (WIESMANN et al., 2000) destaca que esta abordagem pode ser utilizada em serviços não-determinísticos, porém esta abordagem tem um custo de reconfiguração alto, quando a réplica primária falha.

A Figura 3 ilustra um exemplo de Replicação Passiva, no qual um cliente faz uma requisição à réplica primária. Ao receber a requisição, esta réplica executa a ação demandada, e informa às demais que elas devem processar a ação. Ao receber a confirmação da maioria das réplicas, a primária responde ao cliente.

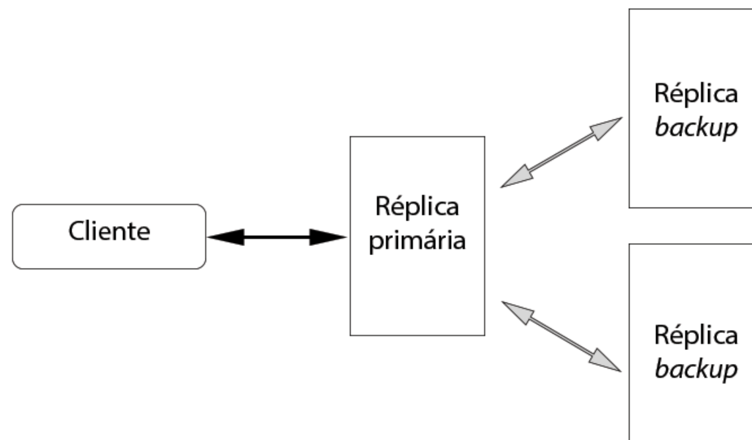

Figura 3 – Representação Replicação Passiva. Fonte: autor

1.2.2 Replicação Ativa

A replicação ativa, também conhecida como Replicação Máquina de Estados (do inglês *State Machine Replication* - SMR) (SCHNEIDER, 1990; LAMPORT, 1978), é uma técnica de replicação descentralizada. Isto é, não há a existência de um único servidor que processe todas as requisições. Seu conceito fundamental é que todas as réplicas recebem e executam uma mesma sequência de requisições dos clientes (WIESMANN et al., 2000). Um dos elementos-chave nesta abordagem é a garantia de que todas as réplicas estejam consistentes entre si, isto é, os mesmos comandos são executados por todas as réplicas, na mesma ordem, de forma determinística, garantindo que todas as réplicas passem pelos mesmos estados (SCHNEIDER, 1990; WIESMANN et al., 2000). Por determinismo, entende-se comandos que independem de fatores que possam variar de réplica para réplica. Por exemplo, uma função que utilize o relógio interno do computador é não-determinística, já que depende do hardware específico daquela réplica. Além disso, as leituras dos relógios pelas diferentes réplicas provavelmente aconteceriam em instantes diferentes.

O principal desafio desta abordagem é garantir a entrega das mensagens de forma ordenada, o que pode ser uma tarefa complexa, devido a falhas nas réplicas, perda de mensagens e o comportamento não-confiável da rede (RENESSE; SCHIPER; SCHNEIDER, 2015). Segundo (SCHNEIDER, 1990), a coordenação entre as réplicas pode ser decomposta em dois requisitos que devem ser cumpridos: Acordo (*Agreement*), no qual todas as réplicas funcionais devem receber todas as requisições, e Ordem (*Order*), todas as réplicas em execução devem receber as mensagens na mesma ordem relativa.

A Ordem pode ser garantida atribuindo identificadores únicos a todas as requisições, e fazendo com que as máquinas executem baseadas neste identificador (SCHNEIDER, 1990), sempre começando pela requisição de menor identificador que ainda não tenha sido executada. Porém, como todas as réplicas precisam receber todas as requisições, a

escalabilidade utilizando este método é limitada: a vazão do sistema não aumenta, conforme novas réplicas vão sendo acrescentadas ao sistema (BEZERRA; PEDONE; RENESSE, 2014).

1.2.3 Abordagens híbridas

Embora a literatura aborde com maior profundidade as técnicas de replicações ativa e passiva, algumas abordagens foram desenvolvidas mesclando características das abordagens anteriormente descritas, tendo como exemplo a replicação semi-ativa e a replicação semi-passiva. A seguir, estas duas variações são discutidas.

Na replicação semi-ativa, projetada por (POWELL; CHÉRèQUE; DRACKLEY, 1991), os autores focaram na questão do determinismo das requisições, e assumiram um modelo síncrono. A diferença desta abordagem em relação à ativa é que essa não requer que os comandos sejam determinísticos. Quando uma requisição não-determinística é feita, uma das réplicas, chamada de líder, trata a requisição e informa às demais réplicas o resultado da operação. Após haver o retorno de um quórum das réplicas, o líder responde ao cliente (WIESMANN et al., 2000).

Já na replicação semi-passiva, algumas características da replicação passiva são mantidas, especificamente:

1. Na ausência de servidores falhos ou suspeita de falha, a requisição é tratada por uma única réplica;
2. O processamento da requisição pode ser não-determinístico.

Ao contrário da replicação passiva, porém, nesta abordagem as requisições são enviadas a todas as réplicas, e todas retornam uma resposta ao cliente. Assim, a falha de uma ou mais réplica fica oculta ao cliente, que não precisa reenviar a requisição após um determinado período, ou saber qual a identidade da réplica primária (POWELL; CHÉRèQUE; DRACKLEY, 1991).

1.3 Protocolos de Ordenação

Um protocolo de ordenação garante que uma série de mensagens sejam entregues na mesma ordem para todas as réplicas em um sistema. Em um SMR, esta condição é imprescindível para que as réplicas permaneçam consistentes entre si. Esta ordem pode ser assegurada utilizando-se difusão atômica ou através de um protocolo de consenso. A seguir, estão descritas estas alternativas.

1.3.1 Difusão atômica

A difusão atômica, do inglês *Atomic Broadcast* (ou *Total Order Broadcast*) (DÉFAGO; SCHIPER; URBáN, 2004), garante que mensagens enviadas a um grupo de processos são entregues a todos os processos corretos na mesma ordem (DÉFAGO; SCHIPER; URBáN, 2004). De um modo mais formal, há a existência de duas primitivas, *TO-broadcast*(m) e *TO-deliver*(m), onde m é uma mensagem qualquer. Além disso, assume-se que toda mensagem possui um identificador único, bem como a informação de quem foi seu emissor. As premissas deste protocolo são as seguintes:

- **Validade** Se um processo *TO-broadcast* uma mensagem m, então eventualmente será feito o *TO-deliver(m)*.
- **Acordo uniforme** Se um processo entrega uma mensagem m, então todos os processos corretos entregarão, eventualmente, a mensagem m.
- **Integridade uniforme** Para qualquer mensagem, cada processo somente entrega uma única vez, e somente se m foi anteriormente difundida (*TO-broadcast*) por algum processo.
- **Ordem total uniforme** Dados dois processos p e q que enviam m e *m'*, p entrega m antes de *m'*, somente se q entrega m antes de *m'*.

Porém, estas propriedades são uniformes, e são aplicadas até mesmo para processos em falha. Assim, segundo estas premissas, um processo faltoso não poderia entregar nenhuma mensagem fora de ordem. Há, entretanto, duas premissas que aplicam-se somente aos processos corretos, não colocando nenhuma restrição aos demais processos (DÉFAGO; SCHIPER; URBáN, 2004). São elas:

- *Acordo* Se um processo correto entrega uma mensagem m, então todos os processos corretos eventualmente entregarão m.
- *Ordem total* Dados dois processos corretos p e q que enviam m e *m'*, p entrega m antes de *m'*, somente se q entrega m antes de *m'*.

Assim, um algoritmo de difusão atômica é uniforme se ele satisfaz acordo e ordem total uniformes, ou não-uniforme, quando não impõe restrições aos processos faltosos.

1.3.2 Paxos

O protocolo Paxos foi projetado por Leslie Lamport, originalmente descrito no artigo “The Part-Time Parliament” (LAMPORT, 1998), no qual é feita uma analogia com

os hipotéticos problemas enfrentados por membros de um parlamento na ilha grega Paxos. Porém, a analogia foi, para muitos, um tanto confusa, de modo que Lamport publicou o artigo “Paxos Made Simple”, no qual ele afirma que o papel de um algoritmo de consenso é garantir que, dentre uma coleção de processos que podem propor valores, apenas um seja escolhido (LAMPORT, 2001). O uso do Paxos repetidas vezes, para obter consenso em uma sequência de valores, como em um *log* replicado, é chamado por muitos autores como MultiPaxos (CHANDRA; GRIESEMER; REDSTONE, 2007a), embora o termo Paxos também possa ser utilizado no contexto de vários valores (BIELY et al., 2012).

O protocolo parte das premissas de que réplicas podem falhar (*crash*) e se recuperar, e que a rede não é confiável, de modo que mensagens podem ser perdidas, mas não corrompidas - o protocolo original não trata falhas bizantinas (LAMPORT, 2001). Toda réplica deve possuir um armazenamento persistente, para que possa, quando se recuperar de uma falha, atualizar-se a partir de outras réplicas (CHANDRA; GRIESEMER; REDSTONE, 2007a).
O Paxos trabalha com três agentes: *proposers*, *acceptors* e *learners*. É possível que uma réplica atue como um destes papéis, ou em mais de um, simultaneamente (LAMPORT, 2001). Esta distinção é, em muitos casos, irrelevante e assume-se que uma réplica exerce os três papéis (BIELY et al., 2012).

O *proposer* tem por função enviar propostas de valores aos *acceptors*, que por sua vez, avaliam se o valor será escolhido ou não. Por fim, o *learner* recebe o valor escolhido por um quórum de *acceptors*, e, uma vez escolhido, este valor não pode ser alterado. O processo de escolha de um valor é dividido em duas fases, usualmente denonimadas de Fase 1 e Fase 2. Internamente, cada fase é subdividida em dois estágios, a e b. O processo começa quando um *proposer* envia uma mensagem do tipo *Prepare*, passando como parâmetro n, um identificador único para cada requisição. O *acceptor* só pode aceitar esta requisição caso ainda não tenha se comprometido com nenhum n maior ou igual aquele informado no *Prepare*. Isto é, ao receber um valor, e caso este ainda não tenha sido ocupado, o *acceptor* compromete-se a recusar qualquer outra requisição com um n inferior ou igual. Neste caso, ele pode simplesmente descartar a mensagem (LAMPORT, 2001). Porém, em muitos sistemas, ao invés de simplesmente descartar a mensagem, a réplica informa ao requisitante que já se comprometeu a um n maior, e qual é o valor deste n. Esta comunicação é a Fase 1, sendo a requisição ao *acceptor* a subfase 1a, e o retorno, 1b. A resposta, caso positiva, contém, além da promessa, a proposta com o maior valor inferior a n que foi aceita, se existir.

Ao obter uma resposta positiva da maioria das réplicas (quórum), a fase 2 pode ser executada. Nesta fase, o *proposer* envia aos *acceptors*, além do número da rodada (n), o valor que será aprendido pelos *learners* (fase 2a). Mais uma vez, os *acceptors* retornam se aceitam ou não o valor proposto, retornando uma mensagem ao *proposer* (fase 2b).

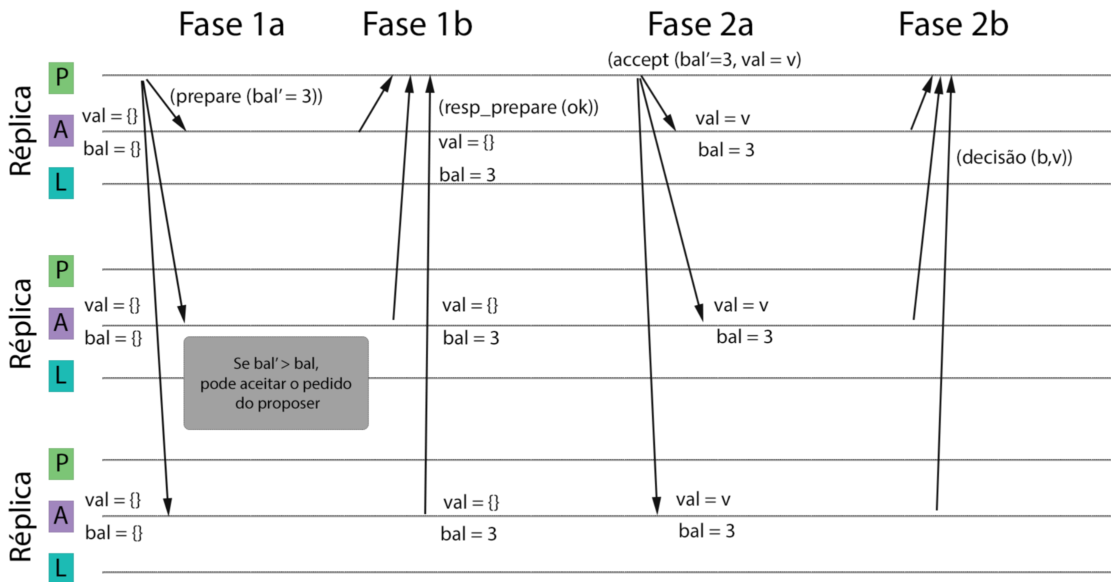

Figura 4 – Divisão das fases no algoritmo Paxos. Fonte: autor

Neste caso, só após o término da fase 2 é que o valor terá sido armazenado, podendo ser processado pelos *learners*. Os *learners* podem ser informados diretamente pelos *acceptors*,ao término da fase 2b, ou pelo *proposer*, gerando uma terceira fase, com a propagação do valor escolhido. O protocolo utiliza uma memória persistente para armazenar os valores aprendidos, bem como a última instância de consenso que a réplica possui.

A Figura 4 mostra essas quatro etapas, em um exemplo utilizando três réplicas, na qual cada réplica executa todos os papéis propostos no Paxos. Neste exemplo, um dos *proposers* começa a Fase 1 enviando as demais réplicas uma mensagem *Prepare* na instância 3. Como estas réplicas ainda não haviam se comprometido com nenhuma instância maior, todas enviam à proponente o consentimento para definir o valor, terminando a Fase 1. Passando à Fase 2, um valor v é enviado pelo *proposer*, sendo aceito pelas réplicas.

1.3.3 Raft

O protocolo Raft (ONGARO; OUSTERHOUT, 2014) tem como principal motivação o desenvolvimento de um protocolo de fácil compreensão, ao contrário do Paxos. Apesar de ser um protocolo consideravelmente novo, tendo sido lançado em 2013, já é utilizado por alguns projetos relevantes, como o etcd (ETCD, 2017) e no RavenDB, banco de dados não relacional para soluções .NET (RACHIS, 2016).
No Raft, há a existência de um nodo que exerce o papel de líder para todas as requisições, enquanto este permanecer funcional. Cada réplica pode estar em um, e somente um, estado em determinado período de tempo: líder, candidato ou seguidor. O líder recebe e processa todas as requisições. No Raft, porém, antes de executar qualquer ação, ele armazena o

comando em um arquivo de *log* e encaminha este *log* às demais réplicas. Só após obter o quórum das réplicas de que o *log* foi replicado (mas ainda não executado), a réplica líder executa seu próprio arquivo, encaminhando a resposta ao cliente, e dando aval para que as demais réplicas executem os *logs* pendentes.

Uma réplica pode estar no estado candidato em duas situações: quando a aplicação está sendo inicializada ou quando a réplica não recebe mais notificações (*heartbeat*) do líder por um determinado intervalo de tempo - usualmente 150 a 300ms. Neste caso, ocorre um processo de escolha de um novo líder, no qual todas as réplicas candidatas votam em si para assumir a liderança, e requerem os votos das demais. No caso de dois ou mais candidatos, é possível que não seja escolhido o líder em uma única rodada, já que pode ocorrer dos votos terem sido divididos, fazendo com que nenhum dos candidatos obtivesse o quórum necessário. Porém, como o intervalo de tempo é aleatório, eventualmente haverá uma janela de tempo grande suficiente para que apenas um candidato consiga os votos necessários à liderança. O último estado possível de uma réplica é o de seguidor, que apenas recebe os *logs* vindo da réplica primária, e, no processo de escolha de um líder, vota na primeira mensagem que receber.

A comunicação entre as réplicas é feita através de RPC (Chamada de Procedimento Remota, do inglês *Remote Procedure Call*), que podem ser de dois tipos: *RequestVote RPC*, utilizada para eleição de uma nova réplica líder, e o *AppendEntries RPC*, que são enviadas do líder para os seguidores, contendo as entradas do *log*, e servem também como *heartbeat*. Neste caso, o parâmetro "*entries*", existente nesta RPC, é vazio (ONGARO; OUSTERHOUT, 2014).

Um *log* é composto por diversas entradas, que são do formato <r,t>, sendo r a requisição e t, o tempo que foi acrescentada ao *log*. "Um termo é o período que começa com a eleição de um candidato e termina com a falta do líder eleito. É representado por um número, e todos os termos sucessivos são maiores que os anteriores" (PINHO et al., 2016). Além disso, a réplica também armazena qual é o índice da entrada do log. Para garantir a consistência, é necessário que todas as réplicas estejam com *logs* equivalentes. A responsabilidade disto é do líder (HOWARD, 2014), e é garantida pelo retorno da Chamada de Procedimento Remoto, que retorna verdadeiro caso tenha sido anexado ao arquivo, ou falso, de outro modo. Cada réplica possui uma variável (*nextindex*) que aponta qual o próximo índice a ser preenchido. Assim, se a réplica recebe um RPC cujo índice seja superior à sua variável *nextindex*, é porque ocorreram perdas, e ele recusa o RPC. O líder vai diminuindo este índice, enviando novas RPC, até que a réplica retorne verdadeiro. A partir dali, ele atualiza uma variável própria, a *matchindex* que indica até qual entrada os dois logs estão consistentes. A seguir, ele vai enviando as entradas, ordenadamente, até que a réplica esteja no mesmo estado (PINHO et al., 2016; ONGARO; OUSTERHOUT, 2014). Segundo (ONGARO; OUSTERHOUT, 2014), caso os seguidores

falhem ou executem vagarosamente, ou se pacotes são perdidos pela rede, o líder reexecuta a *AppendEntriesRPC* indefinidade, mesmo depois de já ter respondido ao cliente, até que, eventualmente, todos os seguidores armazenarão todas as entradas do *log*. Cada entrada pode estar em um único de dois estados: *commited* e *uncommited*. O estado *uncommited* ocorre quando a entrada já foi armazenada no *log*, mas ainda não foi executada, seja porque não recebeu o aval do líder, no caso dos seguidores, ou porque, no caso do líder, ainda não houve uma resposta do quórum de réplicas, para que seja seguro executar a entrada. O *commited* indica aquelas entradas que já foram executadas naquela réplica. Para garantir a propriedade da segurança (*safety*), é necessário que o líder armazene todas as entradas do *log* que foram executadas (as *commited*). Porém, quando o líder falha, algumas garantias extras são necessárias, para evitar que uma réplica defasada seja eleita - isso é vital porque o Raft garante a consistência fazendo com que o líder possa sobrescrever as entradas das demais réplicas. Porém, um líder não pode sobrescrever suas próprias entradas. Assim, se uma réplica defasada for eleita, ela pode sobrescrever entradas que já foram executadas em algumas instâncias, gerando um estado inconsistente. Para tornar-se líder, uma réplica candidata deve contactar, no mínimo, a maioria das réplicas.

Cada entrada executada deve, portanto, estar presente em no mínimo uma destas réplicas. Quando o candidato envia seu pedido, são enviadas também informações sobre seu próprio *log*, para que as demais réplicas possam analisar se o candidato está, no mínimo, tão atualizado quanto ela própria (ONGARO; OUSTERHOUT, 2014). Ou seja, caso haja uma réplica que possua mais entradas executadas em seu *log*, o candidato desiste de sua candidatura. Isso ocorre porque esta réplica que está "mais atualizada", se líder, poderá enviar seus registros aos demais. Mas o líder não pode requisitar registros dos demais, pois este tráfego é unidirecional: do líder aos seguidores.

No aspecto de desempenho, os autores do protocolo afirmam que o desempenho é similar ao do Paxos (ONGARO; OUSTERHOUT, 2014). Apesar da centralização das ações no líder, que pode resultar em gargalos, o desempenho global é garantido pelo baixo número de mensagens enviadas pela rede - como as mensagens partem somente do líder, são poucas as mensagens por entrada no log. Em outros algoritmos, se fazem necessários mais turnos de comunicação para obter o mesmo resultado. Porém, em casos nos quais há muita troca de líderes, um gargalo é observado pelo tempo necessário para escolher um novo líder - no melhor dos casos, um líder é escolhido já na primeira votação. Mas mesmo neste caso, há um atraso, usualmente entre 150 e 300ms. Quando novas votações são necessárias, este tempo impacta no desempenho global.

1.3.4 Viewstamped Replication

O Viewstamped Replication (VR) (OKI; LISKOV, 1988; LISKOV; COWLING, 2012) é um protocolo de ordenação, desenvolvido na mesma época que o Paxos (ver Seção

1.3.2, pág. 21), embora os autores de ambos os trabalhos não tivessem ciência um do outro. O VR foi projetado levando em consideração falhas do tipo *crash*, desconsiderando falhas bizantinas. Assim, para tolerar f falhas, deve-se ter um conjunto de $2f + 1$ réplicas. Ademais, há o conceito de **propriedade da intersecção do quórum**, no qual o quórum de réplicas processando uma determinada etapa do protocolo deve possuir uma intersecção não-vazia com o grupo de réplicas disponíveis para a próxima execução. Assim, em cada etapa, há pelo menos uma réplica que esteve no passo anterior. O protocolo utiliza uma réplica primária para processar as requisições. Ao contrário de outros protocolos, como o Raft, no qual a réplica primária só é substituída em caso de falha, no VR a troca de líder é constante. O protocolo é baseado em visões, sendo que em cada uma, um novo líder é escolhido. Os *backups* ficam monitorando o estado da réplica primária para, em caso de anomalias, seja efetuada uma troca de visão. As réplicas são numeradas a partir de seu endereço IP (*Internet Protocol*), sendo que o menor IP recebe o número um, e assim sucessivamente. O líder é escolhido utilizando uma estratégia *round-robin*[1], começando pela de menor valor (LISKOV; COWLING, 2012).

O cliente, através do Proxy do VR, também armazena informações sobre o estado da aplicação. Entre outras coisas, ele armazena o que ele acredita que é o número da *view*, para localizar o líder naquele período. Isso é possível pois todas as mensagens enviadas aos clientes contém esta informação. Quando o cliente envia uma requisição à réplica, é enviado também o número da requisição (*request-number*), que é único e envios posteriores obrigatoriamente são maiores que o atual. Este número é utilizado tanto pelo cliente quanto réplica: enquanto esta utiliza o número para evitar o processamento de uma mesma requisição duas ou mais vezes, o cliente utiliza para descartar duplicatas vindas da réplica - como o protocolo foi desenvolvido para falhas não-bizantinas, não há preocupação com a veracidade da resposta, assume-se que ela está correta (LISKOV; COWLING, 2012). É enviado, também, o *view-number* que o cliente possui. Quando a réplica recebe a requisição, ela verifica este número, pois só pode processar as requisições cujos *view-numbers* sejam os mesmos que ela possui internamente. Caso o cliente tenha enviado um comando com um número menor que o da réplica, esta simplesmente descarta a mensagem. Por outro lado, caso o cliente possua um número superior, a réplica precisa, antes de realizar qualquer processamento desta requisição, ficar atualizada em relação as demais.

1.4 Técnicas de Programação

Dentre as características desejadas em um *software*, estão a extensibilidade, que pode ser entendida como a capacidade do programa ser extensível para casos de uso não

[1] *Round-robin* é uma estratégia de escalonamento, no qual todos os processos são postos em uma fila circular, sendo executados em sequência. Assim, um processo só voltará a ser executado após todos terem tido sua parcela de tempo.

projetados inicialmente, e a modularidade, que é a construção do *software* em módulos independentes entre si. Algumas técnicas e ferramentas podem ser utilizadas para alcançar estes objetivos, e que serão utilizadas durante o desenvolvimento do arcabouço. A seguir, algumas delas são apresentadas.

1.4.1 Padrões de Projeto

É comum encontrar, no desenvolvimento de *softwares*, casos em que soluções recorrentes podem ser utilizadas. Convencionou-se o nome de Padrões de Projeto a estas abordagens que podem ser aplicadas em alguns contextos já conhecidos. Segundo Guerra (2014b), "um padrão não descreve qualquer solução, mas uma solução que já tenha sido utilizada com sucesso em mais de um contexto".

Os padrões de projeto podem ser classificados da seguinte forma:

- **Padrões de criação:** Tem como objetivo abstrair o processo de instanciação dos objetos. Assim, o objeto que invocar o padrão não precisa se preocupar com detalhes da instanciação da outra classe. São exemplos desta categoria os padrões *Factory*, *Singleton* e *Prototype*.

- **Padrões estruturais:** Estes padrões tratam das associações entre classes e objetos, bem como em melhorar a estrutura das classes. Destacam-se os padrões *Facade* e *Flyweight*.

- **Padrões de comportamento:** Os padrões nesta categoria visam modelar o comportamento das classes e a comunicação entre as classes. São exemplos deste tipo os padrões *Chain of Responsability* e *Strategy*.

1.4.2 Reflexão

Reflexão é o processo no qual um programa pode observar e modificar sua estrutura e comportamento (GUERRA, 2014a), e foi descrito originalmente em 1987, por Pattie Maes (MAES, 1987). Este recurso permite ao programador modificar o comportamento padrão de uma função em tempo de execução. Dentre seus usos principais, está no desenvolvimento de arcabouços de *software*, e na geração de *mocks* [2] durante testes unitários.

Na linguagem Java, as classes que fornecem acesso a esta funcionalidade são encontradas no pacote **java.lang.reflect** (REFLECTION, 2013). Apesar das vantagens oferecidas por esta API, é um recurso a ser usado com cautela, pois apresenta algumas desvantagens. Em primeiro lugar, o uso da Reflexão aumenta a complexidade do *software*, dificultando a depuração do programa. Além disso, é criada uma sobrecarga no sistema, e

[2] Um objeto *mock* é usado para simular comportamentos de objetos reais de forma controlada, usualmente durante testes.

a máquina virtual Java fica impossibilitada de executar determinadas otimizações, como a compilação do *bytecode* de forma mais agressiva. Outra desvantagem a ser considerada é que, por ser utilizada em tempo de execução, certos erros que seriam lançados durante a compilação não são detectados. É o caso, por exemplo, de uma classe externa acessar métodos e atributos privados de outras classes.

1.4.3 Anotações

Uma Anotação é, segundo a Oracle (ANNOTATIONS, 2017), uma forma de metadado que provê dados sobre um programa, não sendo parte do programa em si, nem gerando qualquer efeito sobre o código. Como o nome diz, é feita uma anotação sobre parte do código, podendo ser aplicada aos métodos ou, mais recentemente, aos tipos de dados. Esta anotação é, posteriormente, utilizada pelo compilador, por *softwares* de terceiros ou em tempo de execução (utilizando Reflexão). Em Java, uma anotação é precedida pelo símbolo **@**. Um exemplo de anotações é o **@Override**, introduzido na versão 5 do Java, que indica à máquina virtual que a função anotada sobrescreve a função com mesmo nome da superclasse, no contexto de herança. Outro uso de anotações é em *arcabouços*: o *Hibernate* (HIBERNATE, 2017), ferramenta que faz mapeamento objeto-relacional, utiliza a anotação **@Entity** para indicar que determinada classe deve ser mapeada para uma tabela do banco de dados. Ainda no *Hibernate*, diversas outras anotações são utilizadas, como a **@Id**, que indica que um determinado atributo da classe será a chave primária da tabela.

2 Implementações de Paxos

Dentre as implementações de Paxos de código-aberto, destacam-se aquelas desenvolvidas por grupos de pesquisa de universidades, pois passam por uma revisão de pares, e há artigos que descrevem suas implementações. As bibliotecas analisadas neste trabalho foram o BFT-SMaRt e S-Paxos, desenvolvidas na linguagem Java, LibPaxos, em C, e a OpenReplica, feita em Python. A Tabela 1 apresenta um comparativo entre as diferentes implementações analisadas durante este projeto, indicando a linguagem na qual foi desenvolvida, bem como o modelo de falhas suportado.

Tabela 1 – Comparativo entre as implementações

Biblioteca	Linguagem	Modelo de falha
BFT-SMaRt	Java	Bizantino
S-Paxos	Java	Crash
OpenReplica	Python	Crash
LibPaxos	C	Crash

2.1 BFT-SMaRt

O principal destaque do BFT-SMaRt é o suporte a falhas bizantinas (BESSANI; SOUSA; ALCHIERI, 2014), embora ele possa ser configurado para tolerar *crash*. As réplicas, bem como o modelo de falhas assumido, podem ser modificados em tempo de execução.

Um ponto a ser destacado é que, independentemente da configuração, o BFT-SMaRt requer comunicação confiável (BESSANI; SOUSA; ALCHIERI, 2014), que é obtida através do uso de um autenticador de mensagem sobre o TCP/IP. Além disso, as réplicas possuem uma chave simétrica, produzidas utilizando-se o algoritmo Diffie-Hellman para troca das chaves criptografadas por RSA. No caso de suporte a falhas bizantinas não-maliciosas, nas quais a mensagem pode ser perdida ou chegar ao destinatário corrompida, os clientes não necessitam de chaves, embora esta opção esteja disponível caso necessário.

Para o envio das mensagens, o BFT-SMaRt utiliza o Netty (NETTY, 2017), um arcabouço assíncrono para envio de mensagens TCP e UDP. Como cada mensagem é enviada utilizando uma *thread* Netty, o BFT utiliza todos os núcleos de processamento presentes na máquina para fazer a análise das assinaturas, minimizando o custo extra da criptografia.

Há três tipos de comandos que podem ser executados: *batch*, *unordered* e *ordered*. O primeiro envia um lote de requisições a serem processadas. A diferença entre o segundo

e o terceiro comando é que o *unordered* usualmente refere-se a comandos de leitura, que não alteram nenhum dado e que, portanto, podem ser executados pelo BFT (BESSANI; SOUSA; ALCHIERI, 2014). Já o último, antes de ser executado, precisa ser validado, e adicionado a uma fila de requisições.

2.2 S-Paxos

A motivação do S-Paxos é que nas implementações padrões de Paxos e outros algoritmos baseados em um líder, o gargalo muitas vezes reside no líder. Isso porque, além de atuar como uma réplica normal, o líder precisa ordenar as mensagens, bem como encaminha-las às demais réplicas. Assim, apesar das outras réplicas ainda possuírem recursos disponíveis, o líder atinge seu pico de utilização - seja em uso da CPU ou mesmo da rede (BIELY et al., 2012). Os autores notaram que a Replicação Máquina de Estados realiza três tarefas:

- Disseminação das mensagens: receber as requisições dos clientes, e transmiti-las às demais réplicas.
- Estabelecer ordem: alcançar o consenso sobre a ordem das mensagens.
- Execução das requisições: Executar as requisições na ordem pré-determinada e retornar o resultado aos clientes.

Usualmente essas tarefas são feitas pelo líder. Para minimizar este gargalo, no S-Paxos as mensagens de disseminação podem ser executadas por todas as réplicas, isto é, todas as réplicas podem receber mensagens dos clientes, e elas tratam de encaminhar entre si. Além disso, na ordenação das mensagens, é transmitido somente o id da mensagem, ao invés da mensagem inteira. Por fim, a réplica que recebeu a requisição do cliente é responsável por retornar a resposta (BIELY et al., 2012).

Ao contrário do BFT-SMaRt, o S-Paxos só trata falhas por parada. O envio das mensagens ocorre através de pacotes TCP ou UDP. É possível, ainda, deixar a cargo do programa o envio: para mensagens menores que um valor pré-definido, é escolhido UDP. Passado este valor, o S-Paxos passa a enviar via TCP.

2.3 LibPaxos

A biblioteca LibPaxos permite definir quais papéis (*acceptor*, *proposer*, *learner*) cada réplica irá executar. O envio das mensagens é feito através da biblioteca Libevent (LIBEVENT, 2012), que provê a notificação de eventos de forma assíncrona. As mensagens

são serializadas através da biblioteca MessagePack (MESSAGEPACK, 2013), que utiliza um formato binário para compactação das mensagens.

O projeto é dividido em duas bibliotecas internas: *paxos* e *evpaxos*. A primeira implementa o núcleo do algoritmo, sem estar atrelado a um modo de envio específico - como sockets, ou libevent. Já a biblioteca evpaxos, por sua vez, empacota a lógica envolvendo o envio das mensagens pelo uso do *libevent*.

2.4 OpenReplica

O principal objetivo do OpenReplica (ALTINBUKEN; SIRER, 2012) é desacoplar a lógica do serviço do Paxos. Nesta implementação, o serviço a ser replicado é chamado de "Objeto", e pode ser desenvolvido diretamente no diretório do OpenReplica, ou utilizando uma função via linha de comando ("*concordify*"), quando o código já existir em outra aplicação. Esta função transforma o código Python pré-existente um Objeto, que pode ser replicado.

Neste Objeto ficam os métodos que implementam determinado comportamento. Há, ainda, uma classe *Proxy*, que mapeia os métodos do Objeto para serem replicados. O usuário da aplicação, por sua vez, apenas importa a classe Objeto, instanciando um novo objeto desta classe, e chamando os métodos apropriadamente.

Assim como o BFT-SMaRt, o OpenReplica permite alterar a quantidade e os endereços das réplicas em tempo de execução. Ao contrário das demais bibliotecas, que lidam com endereços IP, o OpenReplica possibilita a integração com o sistema de DNS *Route 53*, da Amazon. Assim, caso hajam trocas de endereços das réplicas, é possível configurá-las diretamente pela interface do *Route 53*, mantendo os mesmos DNS na aplicação, deixando esta configuração transparente ao usuário.

3 Arcabouço para Replicação Máquina de Estados

O objetivo primário do arcabouço é facilitar o desenvolvimento de aplicações utilizando a Replicação Máquina de Estados. Assim, o arcabouço atua como um intermediário entre serviços que devem ser replicados, e as bibliotecas que, de fato, implementam alguma biblioteca de replicação, conforme a Figura 5.

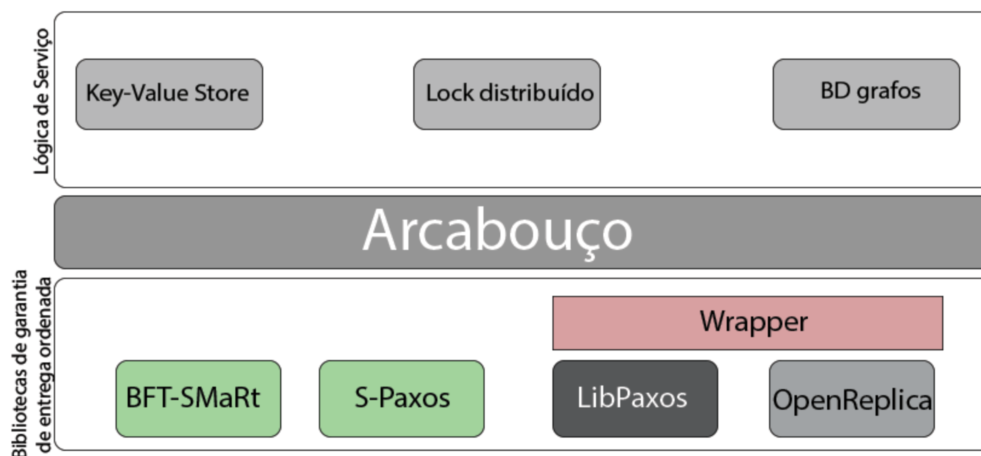

Figura 5 – Modelo conceitual do arcabouço

O arcabouço foi desenvolvido na linguagem Java. A escolha de Java levou em consideração o fato de que duas bibliotecas analisadas foram feitas nesta linguagem, bem como recursos presentes nativamente na linguagem. Foram integradas ao *software* as implementações BFT-SMaRt e S-Paxos.

Ao usuário do arcabouço interessam, principalmente, dois diretórios: *Config* e *Src*. Em *Config* ficam todos os arquivos de configurações, tanto os criados especificamente pelo *software*, quanto aqueles utilizados pelas bibliotecas. Um destes arquivos é o *config.properties*, no qual o usuário informa qual a biblioteca a ser utilizada.

O diretório *Src* contém todos os arquivos .java necessários ao arcabouço. Foi criado um pacote chamado *br.furg.c3.gsde*, bem como alguns pacotes internos nele, como *Impl* e *Services*. Na raiz desse pacote, existem dois arquivos principais, um chamado Réplica e outro Cliente. Cada um destes arquivos possui uma função main, que pode ser utilizado para rodar as instâncias através da linha de comando. O intuito destes arquivos é somente ser um ponto inicial ao programador, mas seu uso não é necessário, uma vez que é possível

chamar as demais classes diretamente, quando, por exemplo, o arcabouço for utilizado em alguma solução que já possua sua classe principal.

Ainda no pacote *br.furg.c3.gsde*, encontra-se o arquivo Setup.java. Este arquivo é responsável pela leitura do arquivo de configuração *config.properties*, bem como atualizar os arquivos de configurações específicos de cada biblioteca, a partir das informações lidas no arquivo de configuração.

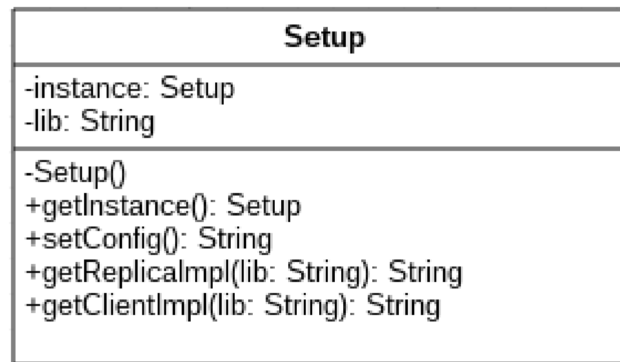

Figura 6 – Diagrama UML da Classe Setup

Por este arquivo ler e escrever em arquivos texto, optou-se por utilizar o padrão de projeto *Singleton*. Este padrão faz com que só exista uma instância de determinada classe durante a execução. Para realizar isto, o método construtor é privado, só podendo ser chamado pela própria classe. Isso impede que outras classes instanciem novos objetos. O diagrama desta classe é mostrado na Figura 6. Nela, há dois atributos privados, sendo um deles uma *String* representando a biblioteca escolhida, e um atributo *instance*, que é um objeto da própria classe. É este atributo que é retornado às demais classes, quando estas requerem uma instância de *Setup*. Por ser privado, é necessário que este atributo seja retornado por um método público, que neste caso é o método *getInstance*. Os outros métodos que podem ser acessados externamente são *getReplicaImpl* e *getClientImpl*, que acessam arquivos de configuração, retornando uma *String* indicando qual o pacote da classe a ser instanciada, a partir de uma biblioteca passada como parâmetro.

3.1 Criando novas réplicas

A inicialização da réplica envolve, de forma abstrata, dois passos: a criação de uma classe genérica na biblioteca, e uma classe que invoque esta classe genérica. Porém, visando a modularidade e extensibilidade do arcabouço, alguns passos extras foram postos, simplificando a integração do arcabouço com bibliotecas vindouras.

As classes internas do arcabouço ficam no pacote *br.furg.c3.gsde.Impl* e são divididas em três arquivos. Primeiramente, foi criada uma interface (ReplicaInterface), que deve

ser implementada pelas classes que fazem a conexão com a biblioteca.

```
// ReplInterface.java

package br.furg.c3.gsde.Impl;

public interface ReplInterface {

void init(int id, String classe);
}
```

A interface contém apenas um método: init, que tem como parâmetros o número da réplica (de 0 a n-1, sendo n o número de réplicas indicadas no arquivo *config.properties*), e a classe que contém o serviço que será replicado.

```
1.package br.furg.c3.gsde.Impl;
2.public class SPaxosReplImpl implements ReplInterface {
3. @Override
4. public void init(int id, String classe) {
5.    GenericReplica genericReplica = new GenericReplica();
6.    try {
7.       genericReplica.init(id, classe);
8.    } catch (IOException | ReplicationException e) {
9.       e.printStackTrace();
10.      }
11.   }
12.}
```

Utilizando a biblioteca S-Paxos como exemplo, foi criado o arquivo *SPaxosReplImpl*, que implementa o método definido pela interface, *init*, como mostrado na linha 4. Já na linha 3, a anotação *@Override* indica à máquina virtual Java que o método deve sobrescrever o método da classe-mãe, que no caso é a interface *ReplInterface*.

Embora a instanciação pudesse ser feita manualmente em uma classe principal, a facilidade ao trocar de biblioteca seria comprometida. Isso porque ao trocar a biblioteca, outra classe deveria ser instanciada. Uma solução paliativa seria utilizar um laço condicional, verificando qual a biblioteca a ser utilizada. A opção utilizada neste arcabouço foi a criação de uma classe de criação, *ReplicaFactory*. Esta classe pega a informação contida em um arquivo de texto sobre qual a classe deve ser instanciada, e, utilizando-se de Reflexão, faz a criação do objeto, retornando uma instância de ReplicaInterface ao requisitante. O código a seguir mostra a implementação desta classe, que retorna, a partir de uma *String*

indicando qual a biblioteca escolhida, uma instância da classe correta.

```
1.public class ReplicaFactory {
2. public static ReplInterface getReplica(String lib){
3.   String classe = Setup.getInstancia().getReplicaImpl(lib);
4. try {
5.   ReplInterface repl = (ReplInterface)
   Class.forName(classe).getConstructor().newInstance();
6.   return repl;
7.   } catch (InstantiationException |  IllegalAccessException |
   InvocationTargetException | NoSuchMethodException | ClassNotFoundException
   e) {
8.      e.printStackTrace();
9.      return null;
10.     }}}
```

Assim, a criação de novas bibliotecas, por parte do arcabouço, envolve apenas a criação de uma classe que implemente *ReplicaInterface*, e faça a correta implementação da biblioteca, e acrescentar o pacote e o nome desta classe no arquivo *modulosReplica.properties*, dentro da pasta Config.

3.2 Criando novos clientes

A estratégia utilizada para modularizar a conexão dos clientes foi a mesma utilizada pelas Réplicas. Dentro do pacote *br.c3.furg.gsde.Impl* foi criada uma interface Java: ClientInterface. Esta interface possui apenas um método, *sendCommand*, que deve ser implementado pelas classes concretas de Cliente.

Figura 7 – Diagrama UML da ClientInterface

O método *sendCommand* possui dois parâmetros, sendo o primeiro o nome do comando que a réplica executará, e o segundo é uma lista de objetos. Este é um requerimento desta versão do arcabouço, pois como o serviço replicado é chamado através de Reflexão, é necessário informar qual o nome do método, bem como os tipos (por exemplo *String*, Double, Integer) dos parâmetros. Toda classe no Java herda *Object* (OBJECT,), menos

os tipos primitivos (*int, float*, *double*, etc.). Assim, os métodos recebem como argumentos objetos da classe *Object*, e uma conversão (*casting*) deve ser feita dentro do método, a cargo do programador.

A instanciação da classe que implementa esta interface é feita através da classe ClientFactory, que obtém a classe concreta a partir de um arquivo, *modulosClient.properties*.

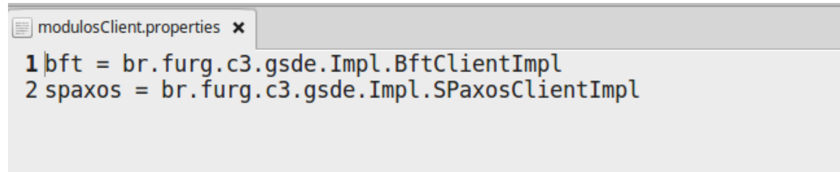

Figura 8 – Arquivo modulosCliente.properties

O arquivo da Figura 8 recebe dois parâmetros que são lidos pela classe *Setup*, o nome da biblioteca e a classe que concretiza a interface *ClientInterface*. Quando a classe *ClientFactory* é invocada, o pacote da classe correta é informado pela classe *Setup*, sendo instanciada através da API de Reflexão.

O código a seguir implementa a interface ClienteInterface, fazendo a conexão com a classe GenericClient, que foi criada dentro do módulo BFT:

```
1.public class BftClientImpl implements ClientInterface {
2. private int id;
3. private GenericClient genericClient;
4. public BftClientImpl(Integer id){
5.    this.id = id;
6.    genericClient = new GenericClient(this.id);}
7. @Override
8. public String sendCommand(String nome, List<Object> args){
9.    String resposta;
10.      try {
11.         ByteArrayOutputStream out = new ByteArrayOutputStream();
12.         ObjectOutputStream dataOut = new ObjectOutputStream(out);
13.         dataOut.writeUTF(nome);
14.         dataOut.writeInt(args.size());
15.         for(Object arg : args){
16.            dataOut.writeObject(arg);
17.         }
18.         byte[]retorno = genericClient.sendCommand(out.toByteArray());
19.         resposta = new String(retorno, StandardCharsets.UTF_8);
20.      } catch (IOException e) {
21.         e.printStackTrace();
```

```
22.         resposta = "ERRO!";
23.      }
24.   return resposta;}}
```

3.2.1 Pacote Services

Dentro do pacote *br.furg.c3.gsde*, há um pacote Services, criado para que o usuário do arcabouço insira o código do serviço a ser replicado. O uso deste pacote não é obrigatório, uma vez que o usuário precisa informar, ao executar o arcabouço, o pacote onde a classe está armazenada. Assim, o usuário poderia criar qualquer pacote dentro da pasta *src*, desde que informando corretamente o caminho. Todavia, este pacote foi criado com o intuito de centralizar esses códigos, bem como ser o local onde serão postos os exemplos de uso do arcabouço.

3.2.2 Anotação Replicar

A anotação Replicar foi criada para que o usuário possa informar quais métodos serão acessíveis aos clientes. Por exemplo, dentro uma classe qualquer, pode haver métodos públicos e privados. Porém, ao usarmos Reflexão, o compilador não pode inferir previamente se determinado método pode ser acessado. Caso um método privado seja acessado deste modo, uma exceção é gerada. Para evitar isso, é possível gerar, no instante que a réplica é iniciada, uma lista contendo todos os métodos que possuírem uma determinada anotação - no caso, **@Replicar**. Assim, ao invés de exceções serem lançados em tempo de execução, é possível informar ao cliente diretamente que tal método não pode ser acessado, sendo ele privado.

Além disso, ao marcarmos determinado método com uma anotação, é possível, através da Reflexão, ter acesso a diversas informações sobre ele, como tipo de retorno, bem como os parâmetros e seus tipos. Assim, é possível fazer uma série de verificações ao receber o comando do cliente, diminuindo a chance do código lançar exceções como *NoSuchMethodException* e *IllegalAccessException*.

3.3 Módulos

Dentro da pasta *modulos*, há, atualmente, duas bibliotecas configuradas: BFT-SMaRt e S-Paxos. Cada uma delas está separada em uma pasta própria, visando uma melhor organização do projeto.

A inserção de novos módulos não deve requerer muitas alterações na biblioteca. Porém, ainda assim, alguns itens são necessários:

- **Criação de uma classe genérica da Réplica**: Tal como existente nos módulos presentes, uma classe que conecte a réplica, bem como leia a classe passada como parâmetro, para poder receber as requisições dos clientes.
- **Criação de uma classe genérica do Cliente**: Como cada biblioteca possui um modo distinto de conectar o cliente e a réplica, optou-se por delegar aos módulos a criação deste cliente, bem como a comunicação entre ele e a réplica. Assim, é necessário a criação desta classe.
- **Arquivos de configuração e geração do JAR**: Os arquivos de configuração devem ficar dentro da pasta *Config*. Assim, é necessário que qualquer referência ao caminho destes arquivos seja atualizada para esta pasta. Outra modificação requerida para o correto funcionamento do arcabouço é que o arquivo executável da implementação, usualmente um arquivo .JAR, fique dentro do diretório *libs*.

3.3.1 Réplica Genérica

A classe genérica da Réplica, que nos módulos já existentes foi chamada de *GenericReplica.java*, tem duas funções principais: criar uma instância do serviço a ser replicado, e receber as requisições dos clientes.

Usualmente, cada réplica recebe um identificador único (*id*), que serve para descobrir seu próprio endereço na rede, bem como os endereços das demais réplicas no sistema. Utilizando somente a biblioteca, sem o arcabouço, este seria o único parâmetro que seria necessário, já que os métodos e o nome da classe replicada já seriam conhecidos anteriormente. Porém, o arcabouço só saberá qual classe é essa em tempo de execução, através da Reflexão. Portanto, o nome completo desta classe (pacote em que a classe está armazenada mais seu próprio nome) deve ser passado por parâmetro. O construtor, assim, recebe dois parâmetros, um inteiro *id* e uma *String* classe: GenericReplica(int id, String classe). Dentro deste construtor a classe do serviço é instanciada, bem como a biblioteca é iniciada.

```
1.public GenericReplica(int id, String classe) throws IllegalAccessException,
    InstantiationException {
2. try {
3.    this.business = Class.forName(classe);
4.    Class annot = Class.forName("br.furg.c3.gsde.Replicar");
5.    try {
6.       Method instance = business.getDeclaredMethod("getInstance");
7.       this.instanciaBusinnes = instance.invoke(null, null);
8.    } catch (NoSuchMethodException e) {
9.       this.instanciaBusinnes = business.newInstance();
10.      } catch (InvocationTargetException e) {
```

```
11.         e.printStackTrace();
12.      }
13.   } catch (ClassNotFoundException e) {
14.   e.printStackTrace();
15.   }
16.   replica = new ServiceReplica(id, this, this);
```

Neste código verifica-se se existe um método *getInstance*. Isso porque, caso a classe utilize o padrão *Singleton*, seu construtor é privado, e caso tentássemos acessá-lo diretamente, via reflexão, uma exceção seria lançada, o que impediria o uso da réplica. Assim, verifica-se primeiro se este método, *getInstance*, existe, na linha 6. Caso não seja encontrado, ele gera uma exceção, linha 8, que é tratada fazendo com que uma nova instância do objeto seja criada, na linha 9.

Os demais métodos da classe *GenericReplica* são dependentes da implementação do SMR em si. Assim, novos módulos devem contemplar essas particularidades na classe Genérica. Todavia, nos dois módulos já existentes, essa integração já foi feita, não sendo necessária nenhuma alteração no arquivo, *a priori*.

3.3.2 Cliente Genérico

Cada módulo deve implementar também uma classe genérica que sirva para enviar os comandos no formato específico daquela biblioteca. Assim, há no mínimo dois métodos que este cliente genérico deve possuir: um construtor, que pode ou não ter parâmetros, a depender da implementação, bem como um método *sendCommand* que recebe ou uma lista de objetos ou um *array de bytes* que será convertido na réplica.

```
1.public class GenericClient {
2. Client client;
3. GenericCommand genericCommand;
4. public GenericClient() throws IOException {
5.   this.client = new Client();
6. this.genericCommand = new GenericCommand();
7. }
8. public byte[] sendCommand(Object... args) throws IOException,
   ReplicationException {
9.   this.client.connect();
10.     byte[] request = this.genericCommand.toByteArray(args);
11.     byte[] response = client.execute(request);
12.     return response;
13.   }
14.}
```

3.4 Resumo

Este capítulo apresentou alguns detalhes de implementação do arcabouço. Há diversos componentes que se relacionam, permitindo uma grande abstração ao usuário final. Um exemplo disso é o uso de Interfaces, que podem ser ampliadas conforme o projeto, passando a ter novos métodos, ou modificando os já existentes. Ademais, como as classes *Factory* têm como retorno a interface, e não a classe concreta, a escolha de uma ou outra biblioteca não impacta no sistema. Exemplificando, caso a classe *ReplicaFactory* retornasse uma implementação concreta, caso houvesse mudança na biblioteca utilizada, este código precisaria ser refatorado. Porém, ao retornarmos a própria interface, essa necessidade não é mais requerida, já que o tipo concreto implementa o contrato definido por ela.

4 Resultados

O foco principal do arcabouço foi tornar a existência das bibliotecas de SMR a mais abstrata possível. Apesar do desenvolvedor ter ciência dos módulos implementados, detalhes de como eles funcionam não são mandatórios para replicar o serviço. Neste aspecto, o arcabouço cumpre esse papel. Em um caso normal de uso, basta ao usuário criar uma classe contendo os métodos acessíveis ao cliente. A seguir, é mostrado uma implementação de banco de dados chave-valor, que armazena os valores em memória, utilizando o arcabouço:

```
1. package br.furg.c3.gsde.Services;
2. %imports otimidos
3. public class SegundoTeste {
4. private Map<Object, Object> bd;
5. public SegundoTeste(){}
6.   this.bd = new ConcurrentHashMap<>();
7. }
8. @Replicar
9. public Object get(Object key){
10.     return bd.get(key);
11.   }
12.   @Replicar
13.   public Object save(Object key, Object value){
14.     bd.put(key, value);
15.     return "ok";
16.   }
17.   @Replicar
18.   public Object delete(Object key){
19.     bd.remove(key);
20.     return "ok";
21.   }
22.}
```

```
1. Setup setup = Setup.getInstancia();
2. String lib = setup.setConfig();
3. int idProcesso = Integer.parseInt(args[1]);
4. ClientInterface client = ClientFactory.getClient(lib, idProcesso);
5. String nome = "save";
```

```
6. List<Object> argumentos = new ArrayList<>();
7. argumentos.add("Um");
8. argumentos.add("Primeiro Teste");
9. String retorno = client.sendCommand(nome, argumentos);
10. System.out.println("Retorno: "+retorno);
```

Os códigos evidenciam que, em um uso padrão do arcabouço, toda a comunicação entre clientes e réplicas é abstraída do desenvolvedor. O serviço a ser replicado compreende apenas os métodos específicos à aplicação. Já do lado do cliente, o envio ocorre através de uma lista de objetos, sem estar atrelado a nenhum modo de envio específico. Estes detalhes ficam nas classes genéricas em cada método, mas de um modo geral, invisíveis ao usuário do arcabouço.

Já sem o uso da biblioteca, além da implementação do serviço, a criação de outras classes era requerida. Por exemplo, no S-Paxos, sem o uso do arcabouço, além do código do banco de dados, uma classe *Service* também deveria ser implementada pelo programador:

```
1. //imports omitidos
2. public class KVService extends SimplifiedService {
3. @Override
4. protected byte[] execute(byte[] bytes) {
5. KVServiceCommand kvcommand = null;
6. try {
7.   kvcommand = new KVServiceCommand(bytes);
8. } catch (IOException e1) {
9. e1.printStackTrace();
10.   }
11.   KVHashMap hashmap = KVHashMap.getInstance();
12.   String comando = kvcommand.getComando();
13.   String key = kvcommand.getKey();
14.   String value = kvcommand.getValue();
15.   String resposta = "";
16.   if (comando.equals("get")){
17.     resposta = hashmap.get(key);
18.   }
19.   else if (comando.equals("save")){
20.     resposta = hashmap.save(key, value);
21.   }
22.     else if (comando.equals("delete")){
23.     resposta = hashmap.delete(key);
24.   }
```

```
25.   else{
26.     resposta = "Comando nao cadastrado";
27.   }
28.   ByteArrayOutputStream byteArrayOutput = new ByteArrayOutputStream();
29.   DataOutputStream dataOutput = new DataOutputStream(byteArrayOutput);
30.   try {
31.     dataOutput.writeUTF(resposta);
32.   } catch (IOException e1) {
33.     e1.printStackTrace();
34.   }
35.   try {
36.     dataOutput.writeUTF(resposta);
37.   } catch (IOException e) {
38.     e.printStackTrace();
39.   }
40.   return byteArrayOutput.toByteArray();
41.   }
}
```

Esse código indica um aspecto que o programador precisa levar em consideração ao utilizar uma implementação qualquer de protocolo, que é o conhecimento *a priori* de quantos parâmetros existem e seus tipos, como evidenciado nas linhas 13, 14 e 15, onde os parâmetros do método são associados a variáveis. Assim, qualquer alteração de parâmetros ou tipo do retorno requer uma refatoração em diversos arquivos.

O arcabouço, desconsiderando os códigos das implementações, que são consideravelmente grandes, totaliza aproximadamente 700 linhas de código, o que torna o arquivo final bastante leve, apesar de funcional.

Porém, por usar massivamente técnicas de Reflexão, que permitiram tornar o arcabouço o mais abrangente possível, uma perda de desempenho deve ser esperada. Não obstante, esta perda de desempenho pode não ser um fator limitador ao uso do arcabouço, principalmente em ambientes com cargas média de tráfego.

Para mensurar qual a degradação estimada, foram realizados alguns testes, com duas classes para teste: uma classe imprimia uma mensagem no terminal, e outra que calculava o número pi com determinada precisão, informada pelo usuário. Em todos os testes, a medição ocorreu iterando-se 100000 instâncias, e medindo o tempo decorrido neste processo, em nanosegundos.

Os testes indicam que para operações pouco custosas, o uso da Reflexão causa uma degradação do desempenho notável, acima de 80%. Todavia, ao testarmos um método que fazia uso intensivo de CPU, essa diferença diminuiu consideravelmente, ficando abaixo de

Tabela 2 – Comparativo Reflexão

Teste	Sem Reflexão	Com Reflexão	Degradação (%)
Imprime Mensagem	745417197	1372167636	84.08
Cálculo PI	1669118928	1930854920	15.68

20%. Como os serviços a serem replicados usualmente envolvem uso de CPU ou de E/S intensos, o uso da Reflexão como recurso da linguagem não é um fator limitador.

Para realizar testes de carga, foram utilizadas seis máquinas com mesma configuração, sendo seus processadores Core i5-4570, rodando a uma frequência de 3.2 GHz, e 8Gb de memória RAM. O sistema operacional utilizado para testes foi o Ubuntu Linux.

Os testes foram realizados replicando um serviço de *Lock*, que possuía somente duas operações: *lock* e *unlock*, conforme código abaixo:

```
//Demais codigos omitidos
@Replicar
public synchronized String Unlock(Object path, Object writer){
   if((Boolean) writer) {
      locks.get((String) path).getReadWriteLock().writeLock().unlock();
   } else {
      locks.get((String) path).getReadWriteLock().readLock().unlock();
   }
   return "Ok";
}
@Replicar
public String Lock(Object path, Object writer){
   if(!locks.containsKey((String) path)){
      locks.put((String) path, new LockModel());
   }
   if((Boolean) writer) {
      locks.get(path).getReadWriteLock().writeLock().lock();
      int newToken = updateToken();
      return (newToken + "");
   } else {
      locks.get((String) path).getReadWriteLock().readLock().lock();
      return (token + "");
   }}
```

Os testes mediram duas métricas: do lado do cliente, a latência da requisição. Isto é, o tempo decorrido entre o envio e a resposta da requisição, en nanosegundos. Já do lado do servidor, foi medida a vazão suportada, isto é, quantas requisições foi capaz de atender, por segundo.

Utilizando-se a biblioteca S-Paxos, os seguintes resultados foram obtidos:

Tabela 3 – Comparativo de desempenho na biblioteca S-Paxos

Nº clientes	**Latência (ns) Sem Arcabouço**	**Vazão (/min) Sem Arcabouço**	**Latência (ns) Com Arcabouço**	**Vazão (/min) Com Arcabouço**
80	195385646	498,407	234547588	465,67
160	274498003	799,723	432459799	512,576
240	291004167	1040,63	580169718	495,741
320	355597410	1170,66	807839002	515,237
400	446535873	1245,65	1086307143	503,196
480	580257582	1255,5	1317969152	483,277

Tabela 4 – Diferença percentual utilizando-se o arcabouço

Nº clientes	**Diferença Latência (%)**	**Diferença Vazão (%)**
80	20,04	-6.57
160	57.55	-35.91
240	99.37	-52.36
320	127.18	-55.99
400	143.27	-59.60
480	127.14	-61.51

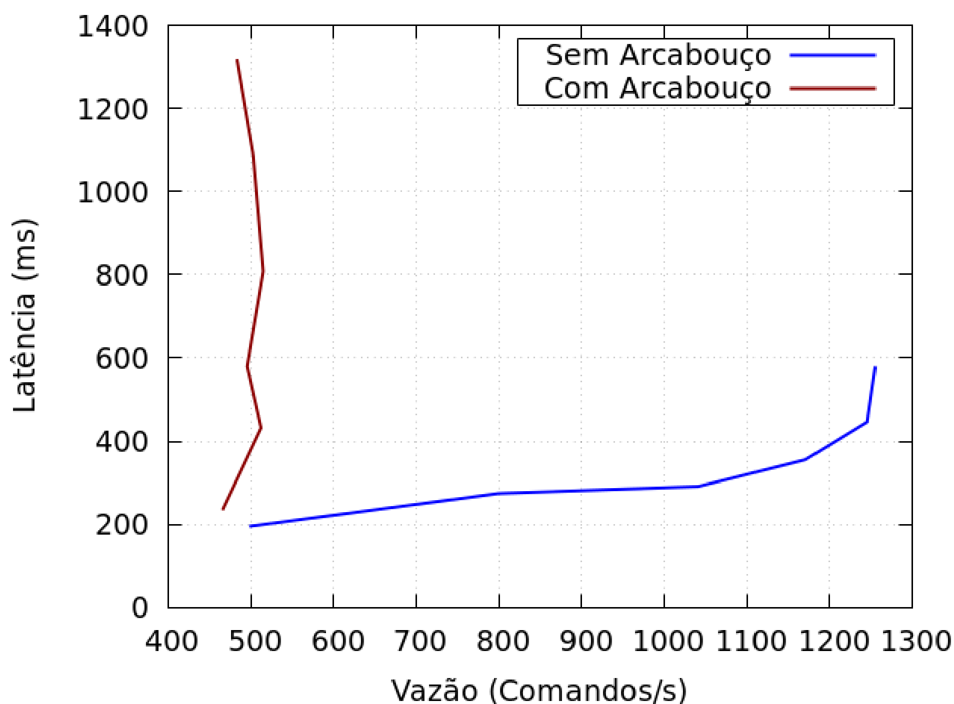

Figura 9 – Gráfico de desempenho do S-Paxos com o arcabouço

Analisando os dados obtidos nas Tabelas 3 e 4, bem como na Figura 9, nota-se uma grande perda de desempenho ocasionada pelo arcabouço. Uma das hipóteses para este resultado é que o uso da Reflexão causa um atraso na execução dos comandos. Todavia,

o uso desta técnica não deveria, por si só, causar tamanha degradação. Outro fator que influencia nisto é o fato de que, utilizando o arcabouço, a classe *Cliente* das bibliotecas recebem uma lista de *Objects* e precisam fazer o envio destes. Utilizando a biblioteca diretamente, o envio dos comandos e parâmetro é feito pelo programador, que já conhece de antemão quais seus tipos, podendo chamar diretamente as funções de envio, por exemplo *DataOutputStream*. Por fim, estes objetos devem ser convertidos para seus tipos originais, como *String*, *Integer* ou outro qualquer, o que não ocorria sem o arcabouço, já que os tipos dos parâmetros eram conhecidos pelo programador que poderia, então, projetar seus métodos para utilizá-los, sem nenhuma conversão de tipos.

Assim, ao escolher utilizar este arcabouço, deve-se considerar que há uma clara perda de desempenho. Em nossos testes, o uso do arcabouço para poucas requisições mostrou um desempenho levemente inferior - a vazão foi prejudicada em pouco menos de 10%, enquanto a latência foi 20% maior, mas ainda assim na casa dos 200ms. Cabe ao usuário mensurar se o custo do arcabouço pode ser suportado pela aplicação, não sendo aplicável em contextos onde há grande número de requisições simultâneas.

5 Considerações Finais e Trabalhos Futuros

O uso de Replicação Máquina de Estados é uma estratégia consolidada, sendo utilizada por diversas empresas, da mesma maneira que no meio acadêmico. Todavia, sua utilização pode não ser tarefa trivial, pois a criação de um *software* que implemente corretamente o protocolo pode ser desafiador, sendo necessários otimizações que podem ou não estar descritas na literatura (CHANDRA; GRIESEMER; REDSTONE, 2007b). Mesmo as implementações de código aberto existentes atualmente requerem ao programador um estudo mínimo sobre seu funcionamento, como quais classes precisam ser estendidas ou implementadas, bem como entender como as classes são relacionadas entre si.

Neste contexto, este projeto vem contribuir criando uma abstração que permite ao programador ignorar detalhes de determinada implementação do protocolo de ordenação. Embora todo o projeto tenha focado no uso do protocolo Paxos, este não é um limitador. Ao tratar cada módulo como uma “caixa preta”, é indiferente ao arcabouço qual o protocolo utilizado, seja ele o Paxos, o Raft ou qualquer protocolo de entrega ordenada de mensagens. As únicas premissas que devem ser seguidas são aquelas descritas no Capítulo 4, que envolvem a criação de uma classe concreta a partir das interfaces definidas. Assim, a modularidade deste projeto torna-se ainda maior, pois além de poder trocar qual a biblioteca será utilizada, é possível ainda trocar o protocolo de ordenação utilizado na Replicação.

Apesar da contribuição deste projeto, há melhorias que podem ser efetuadas em trabalhos futuros, focando-se majoritariamente em dois grandes aspectos: melhoria do desempenho e criação de mecanismos que possibilitem integrar bibliotecas escritas em outras linguagens de programação no arcabouço.

Os testes feitos indicaram que, com o uso de Reflexão e a necessidade de diversas conversões entre parâmetros, o desempenho foi bastante atingido. Com o objetivo de mitigar esta degradação, em uma versão futura do arcabouço, a classe de Reflexão nativa na JVM pode ser substituída. Há alguns projetos que visam estender esta API, tornando-a mais robusta e também mais performática. Um destes projetos é o ReflectASM (REFLECTASM, 2017), que pode ser acoplado ao projeto. Outro aspecto relacionado a Reflexão que pode ser refatorado, é o requerimento de que os argumentos dos métodos replicados sejam do tipo *Object*, sendo necessário um *casting* explícito por parte do usuário.

Por fim, uma possível ampliação deste projeto envolve a criação de bibliotecas intermediárias (*wrappers*) para permitir interoperabilidade com módulos implementados em outras linguagens, como C, Python e Go.

Referências

ALTINBUKEN, D.; SIRER, E. G. *Commodifying replicated state machines with openreplica*. [S.l.], 2012. Citado na página 31.

ANNOTATIONS. 2017. <https://docs.oracle.com/javase/tutorial/java/annotations/>. Acesso em: 14 de ago de 2017. Citado na página 28.

ARIANE. 1996. <http://www-users.math.umn.edu/~arnold/disasters/ariane5rep.html>. Acesso em: 18 de mai de 2017. Citado na página 15.

BESSANI, A.; SOUSA, J. a.; ALCHIERI, E. E. P. State machine replication for the masses with bft-smart. In: *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. Washington, DC, USA: IEEE Computer Society, 2014. (DSN '14), p. 355–362. ISBN 978-1-4799-2233-8. Disponível em: <http://dx.doi.org/10.1109/DSN.2014.43>. Citado 2 vezes nas páginas 29 e 30.

BEZERRA, C. E.; PEDONE, F.; RENESSE, R. V. Scalable state-machine replication. In: IEEE. *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*. [S.l.], 2014. p. 331–342. Citado na página 20.

BIELY, M. et al. S-paxos: Offloading the leader for high throughput state machine replication. In: IEEE. *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on*. [S.l.], 2012. p. 111–120. Citado 2 vezes nas páginas 22 e 30.

BUDHIRAJA, N. et al. Distributed systems (2nd ed.). In: MULLENDER, S. (Ed.). New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1993. cap. The Primary-backup Approach, p. 199–216. ISBN 0-201-62427-3. Disponível em: <http://dl.acm.org/citation.cfm?id=302430.302438>. Citado na página 18.

CHANDRA, T. D.; GRIESEMER, R.; REDSTONE, J. Paxos made live: An engineering perspective. In: *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*. New York, NY, USA: ACM, 2007. (PODC '07), p. 398–407. ISBN 978-1-59593-616-5. Disponível em: <http://doi.acm.org/10.1145/1281100.1281103>. Citado 2 vezes nas páginas 13 e 22.

CHANDRA, T. D.; GRIESEMER, R.; REDSTONE, J. Paxos made live: An engineering perspective. In: *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*. New York, NY, USA: ACM, 2007. (PODC '07), p. 398–407. ISBN 978-1-59593-616-5. Disponível em: <http://doi.acm.org/10.1145/1281100.1281103>. Citado 2 vezes nas páginas 13 e 47.

COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. *Sistemas Distribuídos: Conceitos e Projeto*. 4th. ed. [S.l.]: Artmed Editora, 2007. ISBN 0321263545. Citado 3 vezes nas páginas 16, 17 e 18.

DÉFAGO, X.; SCHIPER, A. Semi-passive replication and lazy consensus. *J. Parallel Distrib. Comput.*, Academic Press, Inc., Orlando, FL, USA, v. 64, n. 12, p. 1380–1398, dez. 2004. ISSN 0743-7315. Disponível em: <http://dx.doi.org/10.1016/j.jpdc.2004.08.006>. Citado na página 18.

DÉFAGO, X.; SCHIPER, A.; URBáN, P. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 36, n. 4, p. 372–421, dez. 2004. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/1041680.1041682>. Citado 2 vezes nas páginas 13 e 21.

ETCD. 2017. <https://github.com/coreos/etcd>. Acesso em: 27 de mai de 2017. Citado na página 23.

GÄRTNER, F. C. Specifications for fault tolerance: A comedy of failures. *TechnicalReport TUD-B S-1998-03 (Oct.), DarmstadtUniversity of Technology, Darmstadt, Germany*, 1998. Citado 3 vezes nas páginas 8, 16 e 17.

GUERRA, E. *Componentes Reutilizáveis em Java com Reflexão e Anotações*. 1. ed. São Paulo: Casa do Código, 2014. ISBN 978-85-66250-50-3. Citado na página 27.

GUERRA, E. *Design Patterns com java*. 1. ed. São Paulo: Casa do Código, 2014. ISBN 978-85-66250-11-4. Citado na página 27.

HIBERNATE. 2017. <http://hibernate.org/>. Acesso em: 8 de set de 2017. Citado na página 28.

HOWARD, H. *ARC: Analysis of Raft Consensus*. [S.l.], 2014. Disponível em: <http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-857.pdf>. Citado na página 24.

LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, ACM, New York, NY, USA, v. 21, n. 7, p. 558–565, jul. 1978. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/359545.359563>. Citado na página 19.

LAMPORT, L. The part-time parliament. *ACM Trans. Comput. Syst.*, ACM, New York, NY, USA, v. 16, n. 2, p. 133–169, maio 1998. ISSN 0734-2071. Disponível em: <http://doi.acm.org/10.1145/279227.279229>. Citado 2 vezes nas páginas 13 e 21.

LAMPORT, L. Paxos made simple. p. 51–58, December 2001. Disponível em: <https://www.microsoft.com/en-us/research/publication/paxos-made-simple/>. Citado 2 vezes nas páginas 13 e 22.

LAMPORT, L.; SHOSTAK, R.; PEASE, M. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, ACM, v. 4, n. 3, p. 382–401, 1982. Citado na página 16.

LIBEVENT. 2012. <https://libevent.org/>. Acesso em: 8 de set de 2017. Citado 2 vezes nas páginas 13 e 30.

LIBPAXOS. 2016. <http://libpaxos.sourceforge.net/paxos_projects.php>. Acesso em: 13 de ago de 2017. Citado na página 13.

LISKOV, B.; COWLING, J. Viewstamped replication revisited. 2012. Citado 2 vezes nas páginas 25 e 26.

MAES, P. Concepts and experiments in computational reflection. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 22, n. 12, p. 147–155, dez. 1987. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/38807.38821>. Citado na página 27.

MESSAGEPACK. 2013. <http://msgpack.org/>. Acesso em: 14 de ago de 2017. Citado na página 31.

NETTY. 2017. <https://netty.io/>. Acesso em: 13 de ago de 2017. Citado na página 29.

OBJECT. <https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html>. Acesso em: 3 de nov de 2017. Citado na página 35.

OKI, B. M.; LISKOV, B. H. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In: *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*. New York, NY, USA: ACM, 1988. (PODC '88), p. 8–17. ISBN 0-89791-277-2. Disponível em: <http://doi.acm.org/10.1145/62546.62549>. Citado na página 25.

ONGARO, D.; OUSTERHOUT, J. In search of an understandable consensus algorithm. In: *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2014. (USENIX ATC'14), p. 305–320. ISBN 978-1-931971-10-2. Disponível em: <http://dl.acm.org/citation.cfm?id=2643634.2643666>. Citado 3 vezes nas páginas 23, 24 e 25.

PINHO, P. R. et al. Replicação de máquina de estado baseada em prioridade com praft. In: *Anais do XXXIV Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*. [S.l.]: UFBA, 2016. Citado na página 24.

POWELL, D.; CHÉRèQUE, M.; DRACKLEY, D. Fault-tolerance in delta-4. *SIGOPS Oper. Syst. Rev.*, ACM, New York, NY, USA, v. 25, n. 2, p. 122–125, abr. 1991. ISSN 0163-5980. Disponível em: <http://doi.acm.org/10.1145/122120.122137>. Citado na página 20.

RACHIS. 2016. <https://ravendb.net/docs/article-page/3.5/csharp/server/scaling-out/clustering/what-is-rachis>. Acesso em: 27 de mai de 2017. Citado na página 23.

RAO, J.; SHEKITA, E. J.; TATA, S. Using paxos to build a scalable, consistent, and highly available datastore. *Proc. VLDB Endow.*, VLDB Endowment, v. 4, n. 4, p. 243–254, jan. 2011. ISSN 2150-8097. Disponível em: <http://dx.doi.org/10.14778/1938545.1938549>. Citado na página 13.

REFLECTASM. 2017. <https://github.com/EsotericSoftware/reflectasm>. Acesso em: 4 de nov de 2017. Citado na página 47.

REFLECTION. 2013. <https://docs.oracle.com/javase/tutorial/reflect/index.html>. Acesso em: 8 de set de 2017. Citado na página 27.

RENESSE, R. V.; SCHIPER, N.; SCHNEIDER, F. B. Vive la différence: Paxos vs. viewstamped replication vs. zab. *IEEE Transactions on Dependable and Secure Computing*, IEEE, v. 12, n. 4, p. 472–484, 2015. Citado na página 19.

SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 1990. Citado 2 vezes nas páginas 13 e 19.

TANENBAUM, A. S.; STEEN, M. v. *Sistemas Distribuídos: Princípios e Paradigmas*. 2ª edição. ed. [S.l.]: Pearson Education do Brasil, 2008. ISBN 9788576051428. Citado na página 15.

WEBER, T. S. Tolerância a falhas: conceitos e exemplos. *Apostila do Programa de Pós-Graduação–Instituto de Informática-UFRGS. Porto Alegre*, 2003. Citado 3 vezes nas páginas 8, 15 e 16.

WIESMANN, M. et al. Understanding replication in databases and distributed systems. In: *Proceedings of the The 20th International Conference on Distributed Computing Systems (ICDCS 2000)*. Washington, DC, USA: IEEE Computer Society, 2000. (ICDCS '00), p. 464–. ISBN 0-7695-0601-1. Disponível em: <http://dl.acm.org/citation.cfm?id=850927.851782>. Citado 4 vezes nas páginas 13, 18, 19 e 20.