UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
CURSO DE GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

Caroline Martins Alves

Uma biblioteca para replicação no RocksDB utilizando comunicação em grupo

Florianópolis
2022

Caroline Martins Alves

Uma biblioteca para replicação no RocksDB utilizando comunicação em grupo

Trabalho de Conclusão do Curso de Graduação em Sistemas de Informação do Centro Tecnológico da Universidade Federal de Santa Catarina como requisito para a obtenção do título de bacharel em Sistemas de Informação.
Orientador: Prof. Odorico Machado Mendizabal, Dr.

Florianópolis
2022

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Martins Alves, Caroline
Uma biblioteca para replicação no RocksDB utilizando comunicação em grupo / Caroline Martins Alves ; orientador, Odorico Machado Mendizabal, 2022.
54 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Sistema de Informação, Florianópolis, 2022.

Inclui referências.

1. Sistema de Informação. 2. Sistemas distribuídos. 3. Replicação. 4. Tolerância a falhas. 5. Comunicação em grupo. I. Machado Mendizabal, Odorico. II. Universidade Federal de Santa Catarina. Graduação em Sistema de Informação. III. Título.

Caroline Martins Alves

Uma biblioteca para replicação no RocksDB utilizando comunicação em grupo

O presente trabalho em nível de bacharelado foi avaliado e aprovado por banca examinadora composta pelos seguintes membros:

Prof. Cristian Koliver, Dr.

Profa. Patricia Della Méa Plentz, Dra.

Coordenação do Programa de Graduação
em Sistemas de Informação

Prof. Odorico Machado Mendizabal, Dr.
Orientador

Florianópolis, 2022.

RESUMO

No cenário de desenvolvimento de sistemas distribuídos há uma série de desafios encontrados, que vão desde a heterogeneidade de servidores que hospedam aplicações, passando por escalabilidade e segurança, indo até o tratamento de falhas. Este trabalho tem como foco o último ponto e, mais precisamente, a replicação. Replicação é uma técnica usada para prover tolerância a falhas, que consiste em manter réplicas de um sistema disponíveis em diferentes servidores. Entretanto, nem todos os sistemas distribuídos implementam essa característica originalmente e, consequentemente, é necessário que o desenvolvedor realize a programação de uma lógica de replicação dentro do sistema para que seja possível utilizá-la. Esse processo pode ser caro, demorado e suscetível a erros, pois implica que o programador possua conhecimentos específicos relacionados a sistemas distribuídos e tolerância a falhas. Visando melhorar e facilitar o uso de replicação em aplicações, esse trabalho propõe o desenvolvimento de uma biblioteca que ofereça replicação para o banco de dados RocksDB de forma transparente para o programador. Com isso, sistemas que já utilizam ou que desejam utilizar o RocksDB de maneira replicada, podem facilmente fazer isso, apenas utilizando esta biblioteca desenvolvida. Para a implementação da biblioteca foi utilizada a ferramenta para comunicação em grupo JGroups.

Palavras-chave: Sistemas distribuídos. Replicação. Tolerância a falhas. Comunicação em grupo.

ABSTRACT

In the distributed systems development scenario, there are a series of challenges encountered, ranging from a heterogeneity of applications, scalability, security, to fault tolerance. This work focuses on the last point and, more precisely, replication. Replication is a technique used to provide fault tolerance, which consists of keeping replicas of a system available on different servers. However, not all distributed systems implement this characteristic and, consequently, the developer is responsible for programming a replication logic within the system so that it can be used. This process can be expensive, time-consuming, and error-prone as it implies the programmer to have specific knowledge related to distributed systems and fault tolerance. Aiming to improve and facilitate the use of replication in applications, this work proposes the development of a library that offers replication to the RocksDB database in a transparent way for the programmer. Thus, systems that already use or want to use RocksDB in a replicated way can easily do so, just importing the developed library. For the implementation of the library was used the JGroups group communication tool.

Keywords: Distributed systems. Replication. Fault tolerance. Group communication.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
DNS	*Domain Name System*
FIFO	*First In First Out*
GbE	*Gigabit Ethernet*
GMS	*Group Membership*
IDE	*Integrated Development Environment*
IP	*Internet Protocol*
RAM	Random Access Memory
SSD	*Solid-State Drive*
TCC	Trabalho de Conclusão de Curso
TCP	*Transmission Control Protocol*
UDP	*User Datagram Protocol*

SUMÁRIO

1 INTRODUÇÃO

Com a evolução tecnológica, muitos serviços que antes só podiam ser executados fisicamente passaram a existir também no mundo digital. Isso deixou as pessoas cada vez mais conectadas e, por vezes, dependendo dos serviços disponíveis na rede. Um vez que utilizam esses sistemas para trabalhar, para comunicar-se com família e amigos, para efetuar transações bancárias, para comércio eletrônico, entre outros. Essa mudança gerou uma grande praticidade aos usuários destes serviços, uma vez que diversas tarefas podem ser realizadas no conforto de suas casas, usando apenas um dispositivo conectado à Internet para acessar diferentes plataformas. Por outro lado, é necessário, também, que os provedores desses serviços garantam a sua disponibilidade e sua funcionalidade.

Quando um desses sistemas fica indisponível, acaba por gerar diversos transtornos e prejuízos para o usuário final, bem como para a empresa provedora. Para exemplificar essa situação, pode-se destacar alguns acontecimentos, como o GitHub que, em fevereiro de 2020, esteve fora do ar diversas vezes, devido a variações inesperadas no carregamento do banco de dados e a problemas na configuração do mesmo (SIMONE, 2020). Em agosto de 2021, o Banco do Brasil ficou fora do ar por cerca de 7 horas e deixou cerca de 54 milhões de clientes sem acesso aos diversos serviços do banco (GLOBO, 2021). No mesmo ano, em outubro, as redes WhatsApp, Facebook, Instagram e Messenger ficaram fora do ar por quase 6 horas, causando um prejuízo de quase 6 bilhões de dólares para a Meta, empresa detentora dessas redes sociais (CNN, 2021).

Nessa perspectiva, os sistemas distribuídos com ênfase em disponibilidade e em escalabilidade vêm ganhando espaço, permitindo o acesso crescente no número de usuários a um mesmo software com rapidez e com estabilidade. De acordo com Tanenbaum e Steen (2007), um sistema distribuído consiste em diversos computadores independentes, apresentados para o usuário como um único sistema coerente. Desse modo, o sistema encontra-se distribuído em diferentes computadores, aproveitando o poder de processamento de cada um destes, mas sem que o usuário perceba isso.

Entretanto, para reduzir os riscos e os impactos no caso de falhas de serviços distribuídos, espera-se que eles cumpram requisitos de sistemas confiáveis, como disponibilidade, confiabilidade, segurança e capacidade de manutenção (VERISSIMO; RODRIGUES, 2001). Para assegurar que essas especificações sejam cumpridas, há várias técnicas de tratamento a falhas que buscam deixar os sistemas distribuídos mais confiáveis, a exemplo da: ocultação de latência, replicação, ocultação de falhas, recuperação de falhas, entre outras.

A replicação, por sua vez, consiste em manter cópias de arquivos, de serviços específicos ou até mesmo de sistemas inteiros, em diferentes servidores, de forma

transparante para os usuários. Isso significa que, no caso de alguma réplica falhar, usuários ainda terão acesso aos dados ou à aplicação, ao conectarem-se a uma das réplicas corretas. Implementar replicação exige o conhecimento de estratégias para coordenação entre as réplicas, uma vez que qualquer alteração feita no estado de uma réplica deve ser reproduzida nas demais (CHARRON-BOST; PEDONE; SCHIPER, 2010). Além disso, é necessário configurar o sistema com um número adequado de réplicas, para garantir o funcionamento correto da técnica e assegurar níveis de disponibilidade adequados. Portanto, a replicação não é facilmente implementada, necessitando que o programador implemente essa característica no sistema explicitamente. Esse processo pode ser custoso, já que requer que o desenvolvedor possua conhecimentos mais avançados em tolerância a falhas.

Visando sanar essa lacuna, este trabalho propõe a criação de uma biblioteca para replicação transparente ao desenvolvedor, baseada em primitivas de comunicação em grupo. Como estudo de caso, foi escolhido o RocksDB (FACEBOOK, 2019; DONG *et al.*, 2017, 2021), um popular banco de dados do tipo chave-valor desenvolvido pelo Meta. O RocksDB não implementa nativamente mecanismos de replicação, então o uso de nossa biblioteca deve prover tolerância a falhas em aplicações que utilizem o banco. Assim, sistemas que desejem possuir um banco de dados em memória replicado poderiam adicionar essa biblioteca ao seu software, sem programar do zero uma estratégia de replicação.

1.1 OBJETIVOS

1.1.1 Objetivo Geral

Este trabalho tem como objetivo principal a criação de uma biblioteca que implementa replicação no banco de dados RocksDB e que pode ser incorporada em aplicações, fazendo assim o uso de replicação de maneira transparente.

1.1.2 Objetivos Específicos

a) Utilizar o paradigma de comunicação em grupo para o desenvolvimento da biblioteca. Para este objetivo, pretende-se explorar a biblioteca JGroups (BAN, 2011);

b) Avaliar o desempenho e custos associados ao uso da biblioteca de replicação. Para isso pretende-se comparar aplicações em execução com e sem o apoio da biblioteca;

c) Realizar a testagem e a validação da biblioteca em ambiente de *testbed*.

1.2 ORGANIZAÇÃO DO TRABALHO

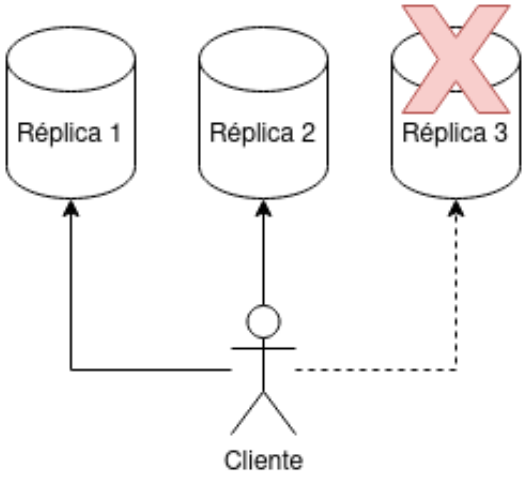
O restante desse trabalho está organizado da seguinte maneira: o Capítulo 2 discute os principais conceitos teóricos e técnicos de ferramentas e paradigmas diretamente relacionados ao desenvolvimento dessa monografia. O Capítulo 3 aborda a biblioteca em si, indo desde as tecnologias usadas, estrutura, funcionamento, utilização, até discussões de aperfeiçoamentos que podem ser incluídos na biblioteca. No Capítulo 4 são apresentados os testes executados a fim de mostrar os resultados decorrentes do uso dela em aplicações, comparando com valores de vazão e latência obtidos com uma aplicação que não utilizou a biblioteca. No Capítulo 5 são exibidos trabalhos relacionados que possuem alguma similaridade com o proposto neste TCC, seja pelo problema que tentam resolver ou por abordarem alguma tecnologia semelhante às utilizadas neste trabalho. No Capítulo 6 são evidenciadas as considerações finais e encaminhamentos para extensão dessa solução em trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Sistemas distribuídos são caracterizados por várias máquinas cooperando entre si, sem que o usuário final saiba da existência dessas diferentes máquinas ou como esse sistema opera internamente. Para isso, essa cooperação entre os diferentes computadores é essencial e, nesse sentido, tem-se que: “Um sistema distribuído é um conjunto de computadores independentes que se apresenta a seus usuários como um sistema único e coerente.” (TANENBAUM; STEEN, 2007).

Para exemplificar esse cenário, pode-se imaginar a seguinte situação: um sistema de transações possui 3 computadores o disponibilizando de maneira íntegra, segura e confiável. Em um dado momento, um desses computadores falha, porém ainda existem 2 computadores operando, conforme ilustrado na Figura 1. Logo, o sistema continua disponível e o usuário pode acessá-lo normalmente. Isso só é possível devido à replicação que concede a esses sistemas a capacidade de continuar disponível mesmo em caso de falhas. Graças a essa técnica de tolerância a falhas, não é necessário saber qual réplica está conectando ou realizar qualquer ajuste (alterar endereço, Internet Protocol (IP), porta, etc.) para continuar realizando as operações. Serviços oferecidos por bibliotecas de comunicação em grupo podem ser utilizados para implementar replicação, facilitando seu desenvolvimento e sua aplicação.

Figura 1 – Funcionamento de um sistema distribuído utilizando replicação



Fonte: A autora (2022).

2.1 COMUNICAÇÃO EM GRUPO

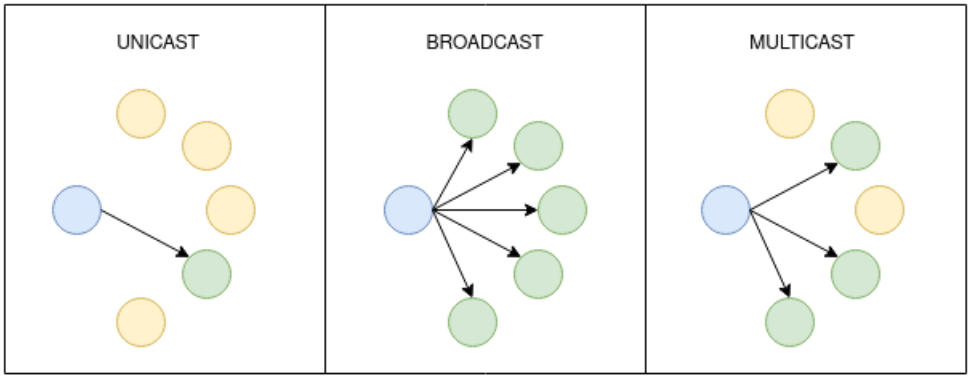
Comunicação em grupo é definida por processos que cooperam a fim de sanar problemas de inconsistência durante a comunicação de sistemas distribuídos. Segundo

(TANENBAUM, 1995), “Um grupo é um conjunto de processos cooperantes (que agem juntos) especificados pelo sistema ou por um usuário” e esses grupos podem receber e disseminar mensagens aos seus processos participantes, bem como enviar mensagens a demais grupos ou aplicações.

Assim, algumas das aplicações possíveis da comunicação em grupo são: realizar disseminação de mídias (a exemplo de um *chat*), além de tarefas mais complexas, como replicação ativa, garantia da entrega das mesmas ações em um contexto de banco de dados distribuído, entre outras.

A comunicação em grupo (BIRMAN, 2005; VERISSIMO; RODRIGUES, 2001) pode ter variações no seu tipo de comunicação, sendo *unicast*, quando as mensagens são trocadas no estilo ponto a ponto, no qual um participante de um grupo envia mensagens a um outro participante desse mesmo grupo, *multicast*, quando as mensagens são trocadas com um grupo de processos e *broadcast*, quando as mensagens são enviadas a todos os participantes do grupo. A Figura 2 apresenta o funcionamento dessas comunicações, ilustrando a comunicação um para um, um para todos e um para muitos, respectivamente. O nodo na cor azul é o remetente e nodo(s) na cor verde é(são) o(s) destinatório(s). Os nodos amarelos não participam da comunicação.

Figura 2 – Tipos de comunicações no paradigma de comunicação em grupo

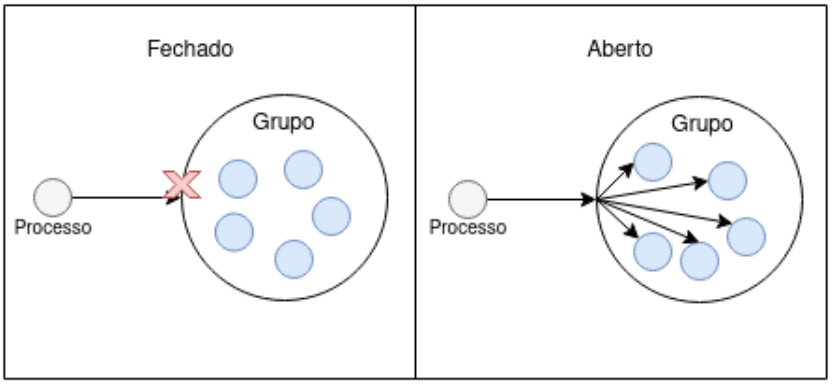


Fonte: Adaptado de (TAHTEC, 2020)

Em relação aos tipos de grupos, eles podem ser abertos ou fechados. Nos grupos abertos, há a interação com o ambiente que está fora do grupo, logo, aplicações externas podem enviar mensagens ao grupo e o grupo pode enviar mensagens a essas aplicações. Já nos grupos fechados, essa interação com o ambiente exterior não existe, havendo relacionamentos apenas dentro do grupo.

A Figura 3 ilustra o funcionamento de um grupo aberto e de um grupo fechado. Na esquerda, há um processo e um grupo fechado com seus membros (nodos azuis), o processo externo tenta emitir uma mensagem ou requisição, mas ela é barrada e não é disseminada entre os membros do grupo. Já na direita, há a ilustração de um

Figura 3 – Tipos de grupos

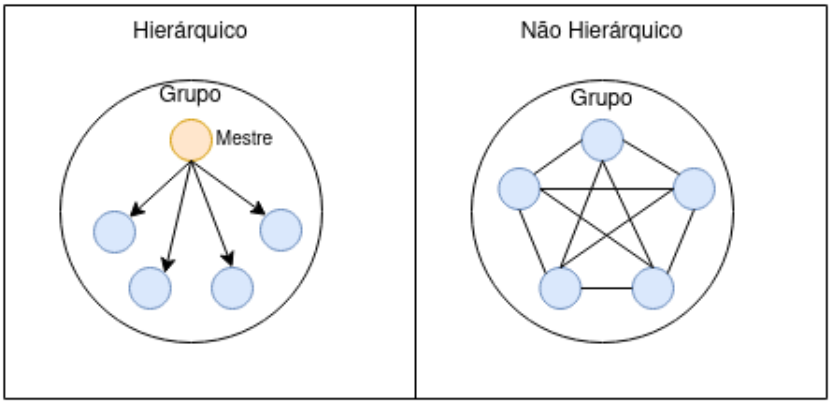


Fonte: A autora (2022).

grupo aberto que, ao receber uma requisição de um processo externo, a dissemina entre todos os membros do grupo.

No quesito hierarquia, os grupos podem ser hierárquicos e não-hierárquicos. No primeiro, os processos do grupo podem desempenhar diferentes papéis, como no caso de haver um nodo coordenador, que controla determinadas ações e, demais nodos, que desempenham outras tarefas. No segundo, todos os participantes desempenham as mesmas funções e atividades no grupo.

Figura 4 – Hierarquia de grupos



Fonte: A autora (2022).

A Figura 4 representa o funcionamento de grupos hierárquicos (lado esquerdo da figura) no qual há um nodo mestre (na cor amarela) e os demais nodos são os trabalhadores (nodos azuis). Nesse cenário, o controle de mensagens fica por conta do mestre. Já no funcionamento de grupos não hierárquicos (lado direito), os nodos são apresentados todos como iguais, sem um coordenador, sendo que as decisões

são tomadas por meio de votação.

Outras características dos grupos são: em relação a sua dinâmica, permitindo que processos juntem-se a grupos existentes ou saiam de um grupo que fazem parte; em relação a sua confiabilidade de comunicação, assegurando que mensagens serão entregues a todos os participantes ou, do contrário, será ignorada, respeitando o preceito de atomicidade - a mensagem chega a todos ou então a nenhum; e em relação aos diferentes tipos de ordenação na entrega de mensagens, como FIFO (*First In First Out*), casual, total, entre outras (BIRMAN, 2005; VERISSIMO; RODRIGUES, 2001).

Assim, este trabalho utilizará comunicação em grupo, onde réplicas do banco de dados serão participantes do grupo, a comunicação será em *broadcast*, já que todos os participantes do grupo receberão as mensagens, fazendo as entregas destas com ordem total para garantir que o estado entre as réplicas será consistente. O grupo é fechado, já que apenas os membros do grupo emitem mensagens. Este funcionamento ficará mais evidente no Capítulo 3, que detalha a estrutura da biblioteca.

2.2 JGROUPS

JGroups (BAN, 2011) é uma biblioteca para comunicação em grupo feita em Java que provê a troca de mensagem de maneira confiável, assegurando a entrega da mensagem a todos os destinatários. Suas principais características são: criar, excluir, entrar e sair de grupos, detecção de falhas em participantes de grupos, remoção de nodos com falhas, envio e recebimento de mensagem em *multicast* e *broadcast*.

Sua incorporação dentro de um código é muito simples, sendo necessário apenas adicionar o .jar da biblioteca no projeto para fazer seu uso. A ferramenta possui com uma documentação detalhada (JGROUPS, 2002), que permite consultar os diversos métodos da biblioteca e entender melhor seu funcionamento e devido uso.

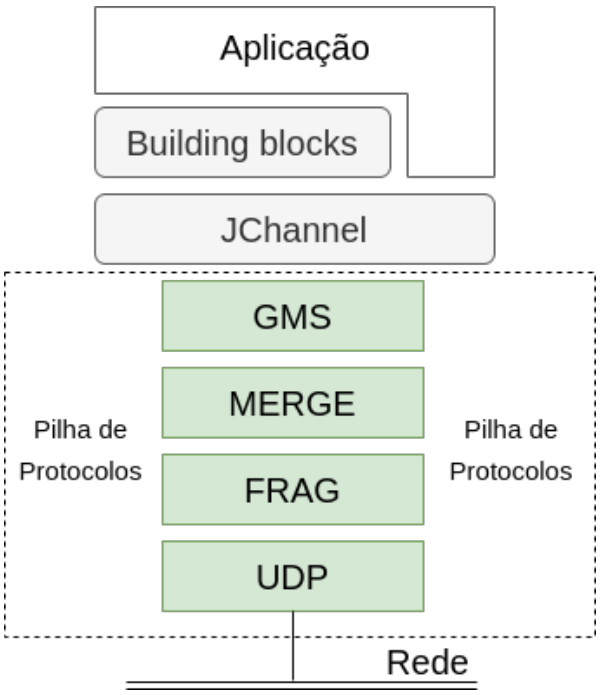
Esse é um *framework* robusto e muito bem consolidado, prova disso são as diversas empresas e soluções de grande porte que o utilizam, a exemplo do *JBoss* que usa o JGroups para implementação de funcionalidade de agrupamento no *JBoss J2EE Application Server*, além de ser utilizado também no *Tomcat JSP* (ABDELLATIF; CECCHET; LACHAIZE, 2004). Ademais, a ferramenta incorpora adequadamente as premissas levantadas pela comunicação em grupo, indo desde a difusão das mensagens, até níveis mais altos como tratamento de falhas dentro do próprio grupo e será um excelente acessório para o desenvolvimento deste TCC. Existem outras bibliotecas disponíveis no mercado que possuem esse mesmo propósito, a exemplo do Atomix (FOUNDATION, 2013), porém optou-se por utilizar o JGroups por ser um produto bem consolidado no mercado, contando com uma documentação bem descrita, o que facilita seu uso.

No JGroups, existem dois conceitos que precisam ser entendidos, o de *cluster*

e o de nodo. *Cluster* é um grupo - processo hospedado em algum *host* e nodo é um membro que conecta-se a um grupo. Diversos nodos podem conectar em um *cluster*. O sistema mantém o controle de nodos que conectam ou desconectam, notificando o *cluster* quando isso ocorre.

A arquitetura do JGroups está organizada em: pilha de protocolos, JChannel e *Building Blocks*, conforme ilustra a Figura 5. A pilha de protocolos possibilita a troca de mensagens, o JChannel é usado pelos desenvolvedores para implementar aplicativos de comunicação em grupo confiáveis. Já os *building blocks* ficam na camada acima do JChannel e proveem um nível maior de abstração, oferecendo implementação de serviços ou estruturas de dados compartilhados entre os membros do grupo. Cada um desses elementos será melhor detalhado na sequência.

Figura 5 – Arquitetura do JGroups



Fonte: Adaptado de (JGROUPS, 2002).

2.2.1 Pilha de Protocolos

A Figura 5 representa essa estrutura de organização do JGroups. Com isso, pode-se visualizar que um canal (JChannel) é conectado a uma pilha de protocolos. A pilha de protocolos é iniciada quando uma aplicação conecta no canal e ao desconectar-se será destruída, liberando espaço para outros recursos. Além disso, ela é configurável, o que permite sua manipulação para atender diferentes necessidades como particularidades de rede.

Essa pilha abrange diversos protocolos, dentre eles os presentes na Figura 5, sendo o *Group Membership* (GMS) que tem a função de gerenciar a composição dos grupos, lançando novos eventos quando há alguma mudança de visão dentro do *cluster*, ou seja, a entrada ou saída de membros do grupo; o *merge* que tem a capacidade de unir um *cluster* que foi separado devido a particionamento de rede; o FRAG que tem seu nome em decorrência da palavra em inglês *fragmentation* e é capaz de segmentar grandes mensagens em porções menores; e o *User Datagram Protocol* (UDP) usado para transporte simples, emitindo mensagens a 1 ou *n* membros do grupo.

Desse modo, quando uma aplicação emite uma mensagem, o canal a transmite na pilha de protocolos, que por sua vez passa para o protocolo mais alto, processando a mensagem e enviando-a para o conseguinte. Logo, a mensagem é passada de protocolo a protocolo, até chegar no responsável pelo transporte e ser enviada na rede. Já o procedimento inverso ocorre para receber uma mensagem. Uma mensagem recebida é entregue para a pilha de protocolos até chegar no canal, que por sua vez aciona um *callback* na aplicação para entregar a mensagem a um procedimento implementado pelo usuário, que irá processá-la.

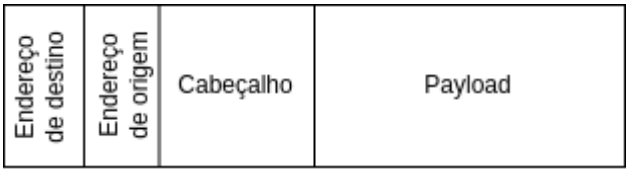
2.2.2 JChannel

Acima da pilha de protocolos está o JChannel que é reponsável por provisionar uma *Application Programming Interface* (API) aos desenvolvedores que utilizem JGroups. APIs fornecem a terceiros um canal para utilizar recursos de uma aplicação, abstraindo detalhes de implementação. No JGroups, a API do JChannel disponibiliza diversos métodos para envio de mensagens, bem como manipulação de canais, que permitem utilizar o paradigma de comunicação em grupo no desenvolvimento de aplicações. Algumas das principais classes disponibilizadas pela API são:

1. JChannel: Responsável por manusear e controlar os grupos. É necessário definir um nome para o JChannel e todos canais de mesmo nome formam um grupo, no qual seus participantes podem enviar e receber mensagens, sendo que os grupos não precisam ser criados previamente, ou seja, se um nodo liga-se a um grupo que não existe, ele é automaticamente criado. Para sair do grupo, um participante pode fazer isso, desconectando-se do canal e devido sua característica de reuso é possível que um membro que desconectou possa conectar novamente caso deseje. Entretanto, cada membro pode conectar-se a um canal apenas, não sendo possível fazer parte de mais de um canal ao mesmo tempo.
2. View: Lista de endereços dos membros de um canal. Quando um membro entra ou sai do grupo essa lista é atualizada e enviada a todos os participantes respeitando a ordem de quem entrou ou saiu.

3. *Message*: Na troca de mensagens entre os membros é enviado um objeto do tipo *Message*. A estrutura desse objeto pode ser visualizada na Figura 6 e consiste em endereço de destino (se for nulo é enviado a todos os membros do grupo), endereço de origem (endereço do emissor da mensagem), cabeçalhos (pode ser adicionada uma lista de cabeçalhos na mensagem, na maioria das vezes isso é utilizado por protocolos) e *payload* (dado de fato).

Figura 6 – Estrutura de uma *message*



Fonte: Adaptado de (JGROUPS, 2002)

4. *Receiver*: Interface que define os *callbacks* que são invocados no lado do *receiver*. Graças a essa classe é possível que cada canal receba as mensagens enviadas.
5. *Address*: Cada membro do grupo possui um endereço que os identifica unicamente o membro. Fornece métodos como de comparação e classificação de endereços.

Existem diversas outras classes que podem ser usadas para diferentes modificações que o desenvolvedor necessite fazer a nível de mensagens, membros, canais e afins. As relatadas acima são as principais, que necessariamente devem estar presentes para criar um canal básico apto a receber e enviar mensagens.

O exemplo de código a seguir foi implementado pela autora, baseado na documentação do JGroups (JGROUPS, 2002) e ilustra a criação de um canal que implementa um recebedor de mensagens e permite a conexão de outros membros dentro do seu *cluster*, estando apto para também enviar mensagens.

```
1  public class JGroupsExample implements Receiver {
2    JChannel channel;
3    public void viewAccepted(View newView) {
4      System.out.println("** view: " + newView);
5    }
6    public void receive(Message msg) {
7        System.out.println(msg.getSrc() + ": " + msg.getObject());
8    }
9    public void start() throws Exception{
10     this.channel = new JChannel().setReceiver(this).connect("Grupo-
         Teste");
```

```
11    }
12    public void sendMessage(Message msg) throws Exception{
13        this.channel.send(msg);
14    }
15 }
```

A entrada em um grupo se dá pela chamada do método *connect* que recebe como parâmetro o nome do *cluster* que se deseja entrar. Note também que a classe JGroupsExample implementa a classe *Receiver* que exige a anotação dos métodos *viewAccepted* e *receive*. O primeiro é responsável por permitir a entrada de novos membros no grupo, graças ao protocolo GMS que emite eventos com atualizações da *view* toda vez que um membro entra ou sai do grupo e o segundo é responsável pelo recebimento de mensagens, graças aos protocolos de transporte que deslocam a mensagem protocolo a protocolo até que ela chegue no canal e possa ser lida. Por fim, como exemplo foi criado o método *sendMessage*, que recebe uma *Message* e a envia no canal a um, vários ou todos os membros (dependendo do que for preenchido no campo de endereço de destino da *Message*), por meio do método *send*.

2.2.3 *BUILDING BLOCKS*

No topo da Figura 5 estão os *building blocks* que ficam na camada acima dos canais e podem ser utilizados ao invés deles, já que são abstrações de nível mais alto que concedem uma API mais sofisticada do que a do *JChannel*. Os blocos de construção, podem criar e usar canais internos ou podem usar um canal existente para sua criação. Além disso, os *building blocks* podem oferecer acesso ao canal e com isso, se o bloco não fornecer alguma funcionalidade, o canal pode ser acessado diretamente. Alguns desses blocos que são fornecidos são:

1. *MessageDispatcher*: Fornece comunicação síncrona e assíncrona, que diferente dos canais, trabalha apenas com assíncrona. Isso é útil para os casos nos quais envia-se mensagem aos membros e deseja-se aguardar até que todos respondam. Essa classe, provê o envio de solicitação bloqueantes/não-bloqueantes e também correlação de respostas. Pode ser usado como cliente, servidor ou, ainda, os dois ao mesmo tempo, caso a aplicação deseje. Para que seja possível entregar requisições no papel de servidor, é necessário implementar o método *handle*. Esse método será chamado sempre que uma requisição for recebida e deve retornar um valor ou lançar uma exceção, o valor retornado será enviado ao remetente;
2. *RpcDispatcher*: Estende o bloco *MessageDispatcher* e permite a chamada de métodos remotos em membros do *cluster* ou que aguarde valores deles. Diferente do *MessageDispatcher* não é necessário implementar o método

handle, apenas é necessário que os métodos a serem chamados sejam assinados de acordo com o padrão criado para esse bloco reconhecê-lo;

3. Bloqueio amplo do *cluster*: O *LockService* pode ser utilizado para realizar bloqueios em todo *cluster*. Sendo que se mais de um membro solicitar determinado bloqueio, o primeiro a realizar a solicitação ficará com o *lock* e o segundo membro não conseguirá adquiri-lo até que o primeiro libere. O bloco *LockService* está associado a um protocolo denominado *CENTRAL_LOCK* ou *CENTRAL_LOCK2* e com isso é possível que o *LockService* "procure"o protocolo e comunique-se com ele por meio de eventos, caso não seja encontrado nenhum dos dois protocolos uma exceção é lançada e o *LockService* não será iniciado.

Esses são apenas alguns dos blocos disponibilizados pelo JGroups que além de muito interessantes, trazem praticidade na resolução de problemas conhecidos. Na documentação da ferramenta (JGROUPS, 2002) há uma seção especial para esse tópico evidenciando todos os blocos disponíveis, além de trazer uma descrição detalhada do sua utilidade, até exemplos práticos de uso.

2.3 ROCKSDB

RocksDB (FACEBOOK, 2019; DONG *et al.*, 2017, 2021) é um banco de dados chave-valor de alto desempenho que faz parte de uma classe de banco de dados, denominada banco de dados embutidos que são bancos totalmente integrados na aplicação. Sua implementação é em C++ e foi feita a partir do LevelDB (GOOGLE, 2021), com o objetivo de ser mais otimizado para realizar armazenamento de maneira rápida e com baixa latência, fazendo um uso eficiente de unidades de armazenamento rápido, como *Solid-State Drive* (SSD) e *Random Access Memory* (RAM).

O banco está estruturado em *log*, nele chaves e valores são apenas cadeias de bytes em tamanho livre, além de estar apto a diferentes cargas de trabalho. Está equipado com mecanismos de armazenamento de banco de dados, cache de dados de aplicativos, cargas incorporadas, entre outras ferramentas. O RocksDB fornece desde operações básicas como abertura e fechamento do banco, como operações mais complexas como mesclagem e filtro de comparação.

Existem outras linguagens em que o banco pode ser utilizado, por meio de bibliotecas disponibilizadas por terceiros, algumas dessas linguagens são: Java, Python, Go, C#, Node.js, PHP, entre outras. Nesse projeto é usada a versão em Java, uma vez que a biblioteca desenvolvida é feita nessa linguagem. Para adicioná-la no projeto, foi feita a importação do arquivo .jar da biblioteca do RocksDB e, a partir disso, já foi possível utilizar o banco na aplicação.

Alguns exemplos de código implementados pela autora, baseados na documen-

tação ilustram como pode ser feito o uso desse banco de dados em java:

- Criando um banco: Primeiramente são inicializadas as opções para o banco (podem ser personalizadas de acordo com a necessidade do desenvolvedor, mas no exemplo são utilizadas as opções padrão). Em seguida cria-se o diretório em que o banco ficará armazenado e por fim o método *open* realiza a abertura desse banco.

```
final Options options = new Options();
options.setCreateIfMissing(true);
this.dbDir = new File("/tmp/rocks-db");
try {
  Files.createDirectories(dbDir.getParentFile().toPath());
  Files.createDirectories(dbDir.getAbsoluteFile().toPath());
  this.db = RocksDB.open(options, this.dbDir.getAbsolutePath());
} catch (IOException | RocksDBException e) {
    e.getStackTrace();
}
```

- Salvando um valor: O método *put* é responsável por guardar os atributos chave-valor dentro do banco. Para isso, basta passar os bytes da chave e do valor para o método e então já serão armazenados.

```
try {
    String key = "1";
    String value = "Teste";
    this.db.put(key.getBytes(), value.getBytes());
} catch (RocksDBException e) {
    e.getStackTrace();
}
```

- Excluindo um valor: O método responsável pela exclusão é o *delete* para realizar a deleção é necessário apenas passar os bytes da chave que se deseja excluir para o método.

```
try {
    String key = "1";
    this.db.delete(key.getBytes());
} catch (RocksDBException e) {
    e.getStackTrace();
}
```

- Retornando um valor: O método responsável por retornar valores é o *get* para realizar a captura de um valor associado a uma chave. É necessário apenas passar os bytes da chave cujo valor deseja ser retornado.

```
try {
    String key = "1";
    byte[] value = this.db.get(key.getBytes());
} catch (RocksDBException e) {
    e.getStackTrace();
}
```

É importante destacar também que “O RocksDB em si não é replicado, mas fornece funções auxiliares para que os usuários implementem seu sistema de replicação sobre o RocksDB [...]” (META, 2020, tradução nossa) [1]. A ausência de estratégia de replicação nativa foi a principal razão para escolhermos o RocksDB como estudo de caso, evitando que o programador tenha que realizar sua versão de replicação para o banco, outros fatores como sua alta performance que resulta em um armazenamento rápido das informações e o uso dele por outras empresas também corroboraram para a escolha desta ferramenta como caso de uso para a solução desenvolvida.

[1] RocksDB itself is not a replicated, but it provides some helper functions to enable users to implement their replication system on top of RocksDB [...]

3 BIBLIOTECA PARA REPLICAÇÃO TRANSPARENTE DO ROCKSDB

Tendo em vista que para utilizar o RocksDB de maneira replicada seria necessário que o desenvolvedor utilizasse as chamadas *Replication Helpers*[2] e criasse seu próprio sistema de replicação, foi desenvolvida essa biblioteca para prover replicação transparente ao RocksDB. Logo, um programador que queira implementar uma solução replicada com RocksDB precisaria, apenas, importar a biblioteca de replicação transparente, sem preocupar-se com a lógica da replicação.

A solução escolhida para implementar replicação de forma transparente foi inserir um *middleware* entre o cliente e o RocksDB, para que as devidas tratativas sejam feitas com o JGroups (BAN, 2011) e, então, os dados sejam replicados. A biblioteca desenvolvida tem um comportamento semelhante a de um *wrapper*, disponibilizando os métodos *open*, *put*, *delete* e *get* do RocksDB e acionando uma sub-rotina a cada operação requisitada, sendo essa sub-rotina responsável pela replicação.

3.1 ESTRUTURA DA FERRAMENTA

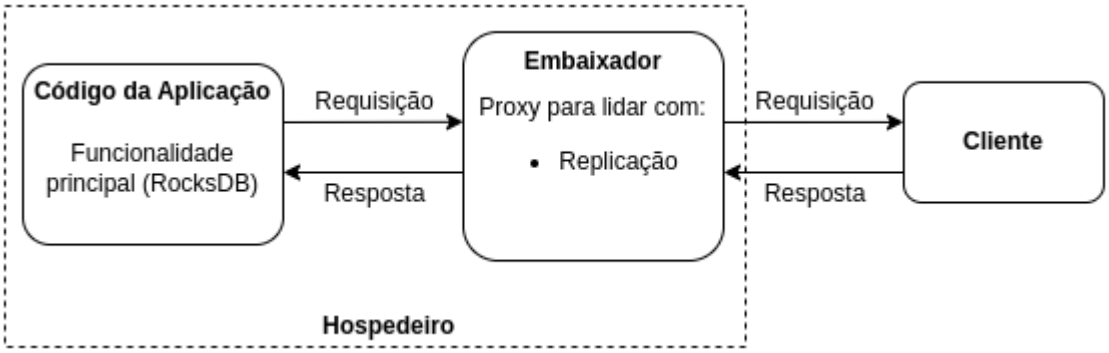
Na implementação foi utilizada a ferramenta de comunicação em grupo JGroups (BAN, 2011) para difusão de mensagens e para garantir todas as tratativas de confiabilidade que um sistema distribuído exige. Optou-se pela linguagem de programação Java, uma vez que a ferramenta JGroups (BAN, 2011) é escrita nela, então o uso dessa linguagem acaba por facilitar o desenvolvimento. Outro recurso escolhido foi o Java Socket por ser um pacote já bem difundido, é bastante prático e fácil de se utilizar (ORACLE, 2014), viabilizando a comunicação do cliente com a biblioteca e vice versa.

Além disso, foi usado um padrão de projeto para a criação deste trabalho, pois esse artifício garante uma maior facilidade para manutenção e para extensão do código na adição de novas *features*. Padrões de projeto são soluções típicas para problemas comuns em projetos de softwares e, para implementação do interceptador, o padrão seguido será o embaixador (MICROSOFT, 2022), que fornece uma maneira transparente de modificar ou de ampliar o processamento tradicional.

A Figura 7 ilustra o padrão de projeto embaixador, que consiste em colocar uma estrutura (Embaixador), que será um interceptador, no mesmo hospedeiro que está localizado o código da aplicação, permitindo ao embaixador controlar roteamento, resiliência e recursos de segurança, dentre outras funcionalidades. Esse funcionamento vai ao encontro da necessidade apresentada pela biblioteca de replicação uma vez que consiste em interceptar uma requisição, realizar um tratamento a mais (replicação) e entregá-la no seu destino (ou nesse caso, destinos).

[2] API disponibilizada pelo RocksDB capaz de prover atualização em réplicas como um componente independente.

Figura 7 – Padrão de projeto embaixador adaptado para funcionamento da biblioteca



Fonte: Adaptado de (MICROSOFT, 2022)

3.2 FUNCIONAMENTO DA BIBLIOTECA

O desenvolvimento da biblioteca considera os quatro métodos principais disponibilizados pela biblioteca do RocksDB, sendo eles: *open*, *put*, *delete* e *get*. Eles permitem, respectivamente, criar uma nova instância do RocksDB, salvar, excluir e retornar dados. É importante frisar que a biblioteca do RocksDB dispõem de outros métodos que realizam desde operações de configuração, como abrir uma nova instância como apenas leitura (*openReadOnly*), até operações com os dados, como limpar todos os dados da tabela de dados (*flush*). Entretanto, nessa solução esses quatro métodos básicos foram implementados, pois são os principais métodos disponibilizados aos clientes para acesso e atualização de dados.

Há uma classe dentro da biblioteca desenvolvida uma classe denominada *ReplicatedRocksDB* que falsamente simula os métodos do RocksDB, mas quando a acionados enviam uma requisição via Socket para uma das réplicas da biblioteca de replicação. Essa réplica, por sua vez, dissemina a mensagem recebida do cliente a todos os membros do grupo de réplicas. A classe *ReplicatedRocksDB* cria um ObjectInputStream que tem informações como: o método que o cliente está tentando acionar, o valor e chave passados ou o nome do diretório para o novo RocksDB que será criado. Após montar esse objeto com as informações necessárias, a classe ainda realiza um *writeObject* concretizando o envio via Socket para uma das réplicas.

A biblioteca, além de ter esse *wrapper* do RocksDB na classe *ReplicatedRocksDB*, possui também um servidor Socket. Desse modo, pode-se levantar *n* réplicas desse servidor que estarão disponíveis para receber a requisição do cliente. A réplica que receberá a solicitação é determinada no lado do cliente, por um arquivo que guarda o endereço IP de todas as réplicas e o envio é feito a alguma dessas réplicas descritas no arquivo. Destaca-se, também, que todo esse processo de levantar as réplicas e preencher o arquivo com os endereço IP é manual e feito previamente ao envio de

uma requisição pelo cliente.

Em relação a interceptação, ela ocorre na classe *ThreadSocket* que é responsável por receber requisições via Socket nas réplicas. Logo, a classe possui um *ObjectInputStream* para desempacotar o objeto enviado pelo cliente e convertê-lo para o formato de mensagens aceito pelo JGroups. Após essa conversão, a réplica que recebeu essa mensagem realiza o envio para todas as réplicas (utilizando o método *send* em *broadcast*). Logo, tem-se um membro de grupo para cada servidor RocksDB disponível.

Por fim, ao chegar a mensagem repassada em cada membro do grupo é feita a identificação de qual método o cliente está invocando, juntamente com seus parâmetros. Cada uma das réplicas do grupo repassa para o RocksDB que é responsável. Assim, se há 3 réplicas da biblioteca, então existem 3 instâncias de serviços do RocksDB. Se o cliente quiser salvar a chave “1” vinculada a *String* “Teste”, esses atributos serão salvos em 3 instâncias do RocksDBs diferentes.

A Figura 8 ilustra o funcionamento esperado do *middleware*, em que o usuário envia uma mensagem com destino a aplicação alvo (RocksDB), a biblioteca intercepta esse envio, capturando a mensagem, fazendo os tratamentos mencionados anteriormente e disseminando-a entre todos os processos participantes do grupo. Por fim, cada participante salva a operação em diferentes cópias da aplicação alvo, concretizando assim a replicação.

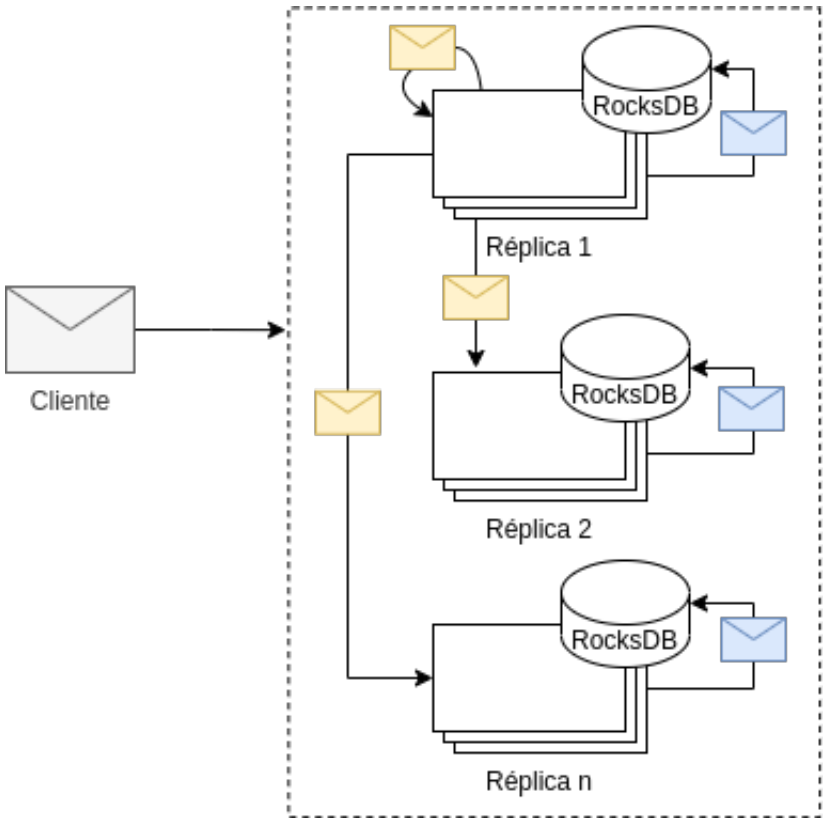
As garantias de atualização e consistência de cada réplica são dadas pela difusão atômica implementada pelo JGroups. Logo, quando uma réplica recebe uma mensagem, ela realizará o envio atômico a todas as demais, então, todas as réplicas recebem ou nenhuma recebe. Além disso, o JGroups garante a ordem total que assegura que as requisições sejam enviadas na mesma ordem. Esses fatores proporcionam a consistência desse modelo, no qual a réplica 1 terá as mesmas informações que a réplica n.

O interceptador será composto por n réplicas da biblioteca, que serão inicializadas e que estarão aguardando a emissão de uma requisição pelo cliente. Assume-se que até f réplicas podem falhar, sendo $f < n$. Assim, o uso do padrão de projeto embaixador no desenvolvimento da biblioteca não cria um ponto único de falha, uma vez que haverá até $n - f$ réplicas ativas, sendo cada réplica responsável por uma instância replicada do RocksDB (FACEBOOK, 2019; DONG *et al.*, 2017, 2021).

3.3 USANDO A BIBLIOTECA EM UM CLIENTE

Para um cliente utilizar o RocksDB de maneira replicada, o primeiro passo é fazer a importação do arquivo .jar da biblioteca de replicação em seu código. Para isso, recomenda-se criar uma pasta denominada *lib* na raiz do projeto e realizar a importação do arquivo de acordo com a *Integrated Development Environment* (IDE)

Figura 8 – Funcionamento da estrutura da biblioteca



Fonte: A autora (2022).

que estiver sendo utilizada.

Em seguida, é necessário inicializar *n* servidores de réplicas, sendo que o recomendado para essa quantidade *n* é $n = f + 1$, no qual *f* é o número máximo tolerado para falhas de réplicas. Para executar os servidores, basta executar o mesmo arquivo .jar adicionado anteriormente, pelo seguinte comando: *$ java -jar replication-library.jar 5001*, sendo que:

- java: Vai executar o código da biblioteca, inicializando o servidor Socket, juntamente com o grupo do JGroups e afins;
- -jar: *Flag* para sinalizar a execução de um arquivo .jar;
- replication-library.jar: Arquivo .jar que tem o código da biblioteca de replicação;
- 5001: Porta em que o servidor Socket vai aguardar requisições. Nesse exemplo escolheu-se a porta 5001 aleatoriamente, porém pode ser qualquer outra que não esteja em uso. Além disso, pode-se usar a mesma porta para todas as réplicas (uma vez que estarão em máquinas diferentes) ou portas diferentes para cada das réplicas.

Por fim, o cliente está apto a utilizar a biblioteca e criar aplicações que tenham

seus dados replicados no RocksDB sem se preocupar em implementar essa característica explicitamente, apenas chamando os métodos normalmente e deixando que a biblioteca de replicação dê essas garantias. A implementação no cliente segue o padrão abaixo:

```
1  import org.replication.ReplicatedRocksDB;
2  public class ExampleApplication {
3      public static void main(String[] args){
4          ReplicatedRocksDB rocksdb = new ReplicatedRocksDB;
5          rocksdb.open("RocksDB-Replicado");
6          String chave = "1234";
7          String valor = "Teste";
8          rocksdb.put(chave.getBytes(), valor.getBytes());
9          byte[] resultado = rocksdb.get(chave.getBytes());
10         rocksdb.delete(chave.getBytes());
11     }
12 }
```

Na linha 1 do código há a importação da biblioteca de replicação e na linha 4 há a instanciação da classe *ReplicatedRocksDB* que fornece os métodos do RocksDB de maneira replicada. Pode-se destacar a preocupação em manter as nomenclaturas dos métodos (linhas 5, 8, 9 e 10) para que o programador ainda tenha a sensação de estar utilizando a biblioteca original do RocksDB, bem como as anotações e valores esperados para cada método. Desse modo, a utilização dos métodos replicados se dá da mesma maneira descrita na Seção 2.3 nos exemplos de codificação do banco. As únicas modificações do código de exemplo para um código de cliente do RocksDB tradicional estão nas linhas 1 e 4.

Esta arquitetura utilizada, mesmo provendo transparência no quesito replicação, ainda possui espaço para melhorias. Uma delas é em relação ao cliente precisar necessariamente conhecer os endereços das réplicas para entregar essas mensagens. Além disso, o interceptador ao qual o cliente está conectado pode falhar. Neste caso, o cliente deve conectar-se a outro interceptador, utilizando o endereço de outra réplica correta. A versão atual da biblioteca não implementa esta reconfiguração. Um serviço de resolução de nomes pode ser utilizado em conjunto com a técnica para permitir que um novo endereço de uma réplica correta seja utilizado pelo cliente no caso de uma perda de conexão com um dos interceptadores.

Em relação a concorrência de clientes também há aprimoramentos a serem feitos uma vez que apenas um dos cliente consegue obter retorno quando há distintos clientes consultando valores com operações de leitura em uma mesma réplica. Esse problema precisa de uma maior investigação para entender o que está ocasionando esse comportamento quando n clientes realizam operações de leitura na mesma ré-

plica.

Além disso, inicialmente foi realizada a implementação dos quatro métodos principais para uso do RockDB (*open*, *put*, *delete* e *get*). Apenas com esses quatro métodos é possível realizar as operações essenciais dentro do RocksDB, porém a biblioteca do RocksDB disponibiliza uma vasta quantidade de operações. Uma versão comercial da biblioteca precisaria implementar *wrappers* para cada uma das operações disponibilizadas pela biblioteca.

Esses pontos de melhoria levantados não impossibilitam o uso da biblioteca, apenas restringem seu alcance. Entretanto, podem ser amenizadas e até mesmo resolvidas em versões posteriores que revisem as limitações da versão atual. O detalhamento da proposta de extensão deste projeto em trabalhos futuros está no Capítulo 6.

4 TESTES E AVALIAÇÕES

Após o desenvolvimento da biblioteca, foram realizadas algumas verificações para analisar qual o custo computacional agregado à solução. É esperado que o uso da biblioteca gere custos em termos de latência, uma vez que, ao invés do cliente entregar a mensagem diretamente ao destino, ela passa primeiramente pelo interceptador. Em termos de vazão, é esperado uma diminuição, ou valores próximos ao da aplicação sem biblioteca.

Foi utilizada a plataforma Emulab (WHITE *et al.*, 2002) para realização dos experimentos. O Emulab é um ambiente de *testbed* que fornece o suporte necessário para alocar nodos e para configurar características de rede em Sistemas Distribuídos em ambiente controlado. Para o teste com a biblioteca, foram alocados *clusters* D430, com a seguinte configuração de *hardware*: 2 processadores de 2.4GHz de 64 bit com 8-cores, 64GB de RAM DDR4, 200GB de SSD e duas placas de rede *Gigabit Ethernet* (GbE). Utilizou-se 4 nodos, sendo um deles o cliente e os demais servidores de réplicas da biblioteca. Já para o teste sem biblioteca fora alocados 2 nodos (de mesma configuração do *clusters* D430), sendo um para o cliente e outro para um servidor Socket que implementa os métodos básicos do RocksDB e a cada requisição feita pelo cliente realiza as operações em uma única cópia do banco de dados.

Os serviços e os clientes foram instrumentados para coletarem a vazão de solicitações processadas, bem como latência na resposta de requisições. Foram elaboradas também execuções sucessivas com acréscimo na carga de trabalho utilizando *threads* e um *loop* que realizava 100.000 iterações. A quantidade de *threads* podia ser personalizada e foi incrementada em potências de 2 para realizar o aumento de carga no sistema.

Os testes foram realizados em duas etapas. A primeira foi para os métodos *put* e *delete*, uma vez que o método *get* estava apresentando problemas ao ser submetido a mais de uma requisição simultânea, conforme mencionado no Capítulo 3. Para escolher o método a ser requisitado havia um sorteio no cliente que determinava qual requisição era enviada, nesse caso utilizou-se uma carga de 50% escrita e 50% exclusão. Tanto o cliente que utilizava a biblioteca de replicação, quanto o cliente que salvava diretamente no servidor do RocksDB, seguiram essa dinâmica, para que os testes fossem padronizados e utilizassem esse mesmo cenário.

Após todas essas configurações serem feitas, foi possível realizar os testes para cada uma das soluções e para os dois diferentes cenários, a quantidade de *threads* inicial usada foi 4 e a cada execução de teste foi dobrada até chegar em 256 *threads*. Os experimentos para cada quantidade de *threads* rodaram por cerca de 90 segundos, sendo interrompidos quando atingiam esse tempo de duração.

Na Tabela 1 são apresentados os valores obtidos de vazão para a solução que

Tabela 1 – Valores de vazão sem o interceptador

Threads	**Vazão (kComandos/s)**
4	487,83
8	878,57
16	1.640,81
32	3.076,41
64	6.313,78
128	12.622,28
256	25.073,13

Fonte: A autora (2022).

não faz uso da biblioteca de replicação. Pode-se perceber um aumento de vazão que acompanha a quantidade de *threads*, não chegando a saturação do sistema.

Tabela 2 – Valores de vazão utilizando o interceptador

Threads	**Vazão (kComandos/s)**
4	491,94
8	885,47
16	1.673,27
32	3.260,05
64	6.427,41
128	12.954,18
256	26.225,43

Fonte: A autora (2022).

Já na Tabela 2 estão os valores obtidos com o cliente utilizando a biblioteca de replicação. Pode-se perceber que o mesmo aumento acontece com a vazão, não saturando o sistema também. Vale ainda ressaltar que a vazão para o cenário com o interceptador foi um pouco maior do que sem, o esperado é que para vazão no teste com o interceptador ela se mantivesse com valores iguais ou abaixo para a vazão no teste sem interceptador. Por exemplo, com 4 *threads* a solução sem replicação obteve 487,83 kComandos/s enquanto a solução com replicação obteve 491,94 kComandos/s. Para versões futuras é importante verificar o porquê deste acontecimento.

A segunda etapa foi para medir a latência na espera da resposta do servidor, utilizando o método *get*. Para isso, utilizou-se uma *thread* com um único cliente, tanto com, quanto sem o interceptador. Inicialmente, nesse teste é feito um pré-carregamento, inserindo cerca de 100.000 valores no banco, para que valores sejam retornados ao invocar o *get*.

Nos dados de latência, os valores obtidos estão na Tabela 3. Foi possível verifi-

Tabela 3 – Valores de latência com e sem o interceptador

Threads	**Latência (ms) Com Intercep.**	**Latência (ms) Sem Intercep.**
1	1,051633298	1,003664449

Fonte: A autora (2022).

car um aumento de 0,05 ms quando utilizado o interceptador para realizar as requisições de *get*. Esse aumento era previsto, entretanto, esperava-se um aumento maior e mais significativo.

Como não haviam os valores de latência para diferentes quantidades de threads, não foi possível gerar um gráfico que descreva a latência em relação a vazão com o acréscimo de carga. Além disso, o ponto de saturação não foi atingido para os testes realizados. Outro fator que pode ter influenciado na não observação da saturação do sistema é os testes terem sido executados com apenas uma máquina cliente. Como havia apenas um cliente conectado no sistema isso pode não ter sido suficiente para estressar o sistema.

Outra análise que seria interessante é submeter o sistema a diferentes porcentagens para cada método. Para esses testes foi utilizada uma carga de 50% escrita e 50% exclusão, mas a instrumentação dos testes foi feita pensando em permitir diferentes porcentagens.

Assim, com os testes realizados e os valores obtidos é possível notar que a vazão teve um aumento quando utilizado o interceptador, o que sinaliza um cenário positivo, uma vez que era esperado uma valor menor ou igual ao da vazão sem interceptador. Para o valor de latência houve um aumento muito pequeno de apenas 0,05 ms. Vale ressaltar que os testes foram executados com apenas um cliente e fazendo acréscimo de carga com *threads*, mas representaram um contexto proveitoso, no qual o uso dessa camada de replicação não apresentou um gasto computacional muito elevado.

5 TRABALHOS RELACIONADOS

Neste capítulo serão apresentados alguns trabalhos que possuem correlação com esta monografia, seja com relação às ferramentas utilizadas, especialmente o JGroups ou aos objetivos de prover replicação de forma transparente. Nesse sentido, os dois primeiros trabalhos apontam formas de implementar essa técnica de tolerância a falhas e o último tece avaliações sobre a ferramenta JGroups em *clusters* de aplicações J2EE.

5.1 BIBLIOTECA PARA REPLICAÇÃO TRANSPARENTE DE SERVIÇOS

Em (PEREIRA *et al.*, 2019), é apresentada uma estratégia para replicar trechos de código de um sistema, utilizando a abordagem de Replicação de Máquina de Estados (RME), por meio do protocolo de consenso Paxos (LAMPORT, 2001). Assim, para que o programador determine que certo método deve ser replicado, é necessário que utilize a anotação *@SmrMethod* (com seus devidos parâmetros), logo acima da assinatura do método. A ferramenta identificará, com base nas anotações de código, quais métodos devem ser replicados.

Desse modo, o serviço de replicação oferecido pela biblioteca aumenta a disponibilidade ao custo de uma perda de desempenho. Além disso, foi observada uma perda de vazão ao utilizá-la, como já era esperado. Houve também um aumento da latência ocasionado pelo uso do protocolo Paxos (LAMPORT, 2001) e não da biblioteca em si.

Essa ferramenta possui um intuito muito parecido com o proposto neste trabalho, uma vez que os dois tratam de replicação em sistemas distribuídos de maneira transparente. Entretanto, a biblioteca de replicação de serviços exige que o programador anote os métodos que devem ser replicados, necessitando de uma intervenção do desenvolvedor e uma alteração da aplicação alvo para utilizá-la. Já nossa proposta visa integrar um serviço de replicação na forma de um interceptador, capturando o tráfego de requisições entre cliente e RocksDB.

5.2 *TRANSPARENT REPLICATION USING METAPROGRAMMING IN CYAN*

A proposta feita por Ugliara, Vieira e Oliveira Guimarães (2017) apresenta como usar a estrutura de meta programação da linguagem Cyan em prol de desenvolver softwares replicados e tolerantes a falha por meio de anotações simples, sem que o programador precise saber detalhes profundos de replicação. Isso pode ser feito por meio de meta objetos que podem ser anexados a métodos modificadores.

O artigo mostra como fazer essa implementação, gerando e validando o código de integração que usa a estrutura de Treplica. Os autores também apresentam como

a ferramenta faz verificações em relação a determinismo, na qual o desenvolvedor é alertado caso exista alguma operação inconsistente ou método modificador. Como trabalhos futuros, os autores pretendem criar um mecanismo capaz de substituir essas operações não determinísticas por operações determinísticas equivalentes.

Assim como o trabalho apresentado em (PEREIRA *et al.*, 2019), os autores propõem uma maneira facilitada de implementar replicação em sistemas. O artigo ainda exige que uma linguagem de programação (Cyan) específica seja utilizada para isso e da mesma maneira, a biblioteca de replicação desenvolvida necessita de uma linguagem específica para funcionar (Java). Para ambos trabalhos esse fator pode ser limitador, uma vez que vincula as soluções a linguagens de programação.

5.3 *EVALUATION OF A GROUP COMMUNICATION MIDDLEWARE FOR CLUSTERED J2EE APPLICATION SERVERS*

Este artigo apresenta uma avaliação de desempenho do *middleware* JGroups (BAN, 2011), tanto em relação ao servidor de aplicações quanto em relação ao *middleware* em si, considerando questões como escalabilidade e performance. Para isso, os autores Abdellatif, Cecchet e Lachaize (2004) avaliaram a configuração de JGroups (BAN, 2011) usada em servidores Tomcat JSP ou JBoss J2EE. O artigo também compara a ferramenta com diferentes configurações para tecnologias de rede, pilhas de protocolo e tamanhos de *clusters*.

O trabalho apresenta que a taxa de transferência usando comunicações $1 : n$ mostra uma melhor escalabilidade com o protocolo UDP em relação ao TCP nos *clusters* com mais de 4 nodos. Entretanto, os *clusters* J2EE utilizam comunicações $n : n$ e nesse cenário o TCP tem desempenho melhor do que o UDP. Já a largura de banda não teve impacto significativo nos testes. Outro destaque importante é que para melhorar o desempenho e escalabilidade dos servidores, os autores sugerem utilizar configurações baseadas em TCP na pilha padrão do JGroups (BAN, 2011) que utiliza UDP.

Assim, esse trabalho tem relação direta com o que foi desenvolvido ao longo dessa monografia, uma vez que JGroups (BAN, 2011) foi a ferramenta principal para a implementação da biblioteca que fornece replicação de maneira transparente. Logo, é importante entender com quais configurações o *middleware* tem melhor desempenho, em busca de possíveis melhorias ou otimizações que futuramente possam ser feitas.

6 CONSIDERAÇÕES FINAIS

O presente trabalho realizou o desenvolvimento de uma biblioteca de replicação voltada para aplicações que desejem utilizar o banco de dados RocksDB de maneira replicada. Desse modo, será possível que desenvolvedores implementem aplicações que terão seus dados replicados, mas sem precisar preocupar-se com a lógica dessa técnica de tolerância a falhas.

Para implementação deste interceptador, camada que intercepta as requisições do cliente com destino ao RocksDB e realiza a replicação dessas solicitações, foi necessário um aprofundamento em conceitos de Sistemas Distribuídos e tolerância a falhas, além de entrar em contato e conhecer outras ferramentas, como o próprio RocksDB e o JGroups que foi utilizado para prover comunicação em grupo entre as réplicas, facilitando a criação da biblioteca.

No Capítulo 3 ainda é explicitado o uso previsto para essa biblioteca, demostrando desde quais comandos o desenvolvedor teria que rodar para criar as réplicas, até um exemplo de codificação correta para utilização dos métodos do RocksDB replicado.

Por fim, a solução criada foi submetida a testes utilizando acréscimo no número de *threads* utilizadas a cada execução, adicionando carga a fim de estressar uma aplicação de teste que fazia uso da biblioteca e compará-la com outra que não a utilizava. Entretanto, os resultados obtidos não chegam em um ponto de saturação, o que dificulta a análise do uso desta abordagem de replicação transparente.

6.1 TRABALHOS FUTUROS

Esta versão ainda pode ser aprimorada, adicionando novas funcionalidades e refinando pontos de melhoria destacados ao final dos Capítulos 3 e 4. A partir disso, surgem possíveis extensões que podem ser feitas, como:

1. Automatizar processo de criação das réplicas: Para criação das réplicas é necessário que o desenvolvedor as inicie explicitamente de acordo com as orientações na Seção 3.3. Esse processo poderia ser automatizado com um arquivo *script* que inicializa as réplicas;
2. Melhorar transparência do cliente no conhecimento das réplicas: Conforme mencionado no Capítulo 3, é preciso que o cliente possua conhecimento dos endereços das réplicas ativas para estabelecer a comunicação inicial e também em caso de falha, saber quais outras réplicas estão disponíveis para receberem requisições. Uma alternativa seria um arquivo de configuração que mantêm registro das réplicas ativas e é repassado ao cliente ou então o uso de um DNS para as réplicas resolveria esta questão;

3. Implementar outros métodos do RocksDB: Atualmente, os métodos *get*, *put*, *open* e *delete* estão disponíveis para uso na biblioteca desenvolvida. Entretanto, o RocksDB disponibiliza uma série de outros métodos. A implementação de outras funções deixaria a solução com suporte a uma gama maior de operações;
4. Aprimorar o suporte a *n* clientes: Apesar de possuir um servidor Socket apto a aceitar *n* clientes, a solução enfrenta alguns problemas de concorrência quando mais de um cliente solicita o retorno de valor de alguma chave na mesma réplica. Este problema de concorrência com o uso da operação *get* precisa ser investigado;
5. Aprofundar testes: O ponto de saturação não foi atingido nos testes executados por essa versão. A sugestão é explorar mais essa questão dos testes, utilizando mais do que um cliente, que não foi possível nessa versão devido aos problemas discutidos ao longo do trabalho;
6. Otimizar solução: Com um aprofundamento dos testes será possível ter uma leitura melhor de possíveis lacunas de desempenho geradas pela biblioteca e mapear quais melhorias podem ser feitas para aprimorar a performance do interceptador;
7. Possibilidade de generalização: Avaliar a viabilidade de generalizar a biblioteca utilizando outras técnicas de programação como reflexão e abstração para que então seja possível utilizá-la em diferentes contextos. Seria muito prático ter uma biblioteca de replicação genérica, que possa atender diferentes aplicações alvo, não apenas o RocksDB.

REFERÊNCIAS

ABDELLATIF, Takoua; CECCHET, Emmanuel; LACHAIZE, Renaud. Evaluation of a group communication middleware for clustered J2EE application servers. *In*: SPRINGER. OTM Confederated International Conferences "On the Move to Meaningful Internet Systems". [*S.l.*: *s.n.*], 2004. P. 1571–1589.

BAN, Bela. **Reliable Multicasting with the JGroups Toolkit**. [*S.l.*: *s.n.*], 2011. `http://www.jgroups.org/manual/html_single/`. [Online; acessado em 15/12/2021].

BIRMAN, Kenneth P. **Reliable Distributed Systems**. [*S.l.*]: Springer New York, NY, 2005.

CHARRON-BOST, Bernadette; PEDONE, Fernando; SCHIPER, Andre. **Replication: Theory and Practice**. [*S.l.*]: Springer, 2010. v. 5959.

CNN. **Mark Zuckerberg perde quase US$ 6 bi em dia de falhas e denúncias ao Facebook**. [*S.l.*: *s.n.*], 2021. https://www.cnnbrasil.com.br/business/mark-zuckerberg-perde-quase-us-6-bi-em-dia-de-falhas-e-denuncias-ao-facebook/. [Online; acessado em 04/03/2022].

DONG, Siying; CALLAGHAN, Mark; GALANIS, Leonidas; BORTHAKUR, Dhruba; SAVOR, Tony; STRUM, Michael. Optimizing Space Amplification in RocksDB. *In*: CIDR. [*S.l.*: *s.n.*], 2017. v. 3, p. 3.

DONG, Siying; KRYCZKA, Andrew; JIN, Yanqin; STUMM, Michael. RocksDB: Evolution of Development Priorities in a Key-Value Store Serving Large-Scale Applications. **ACM Trans. Storage**, Association for Computing Machinery, New York, NY, USA, v. 17, n. 4, out. 2021. ISSN 1553-3077.

FACEBOOK. **RocksDB: A Persistent Key-Value Store for Fast Storage Environments**. [*S.l.*: *s.n.*], 2019. [Online; acessado em 25/07/2022]. Disponível em: `http://rocksdb.org/`.

FOUNDATION, Open Networking. **Atomix Framework**. [*S.l.*: *s.n.*], 2013. https://atomix.io/. [Online; acessado em 14/11/2022].

GLOBO, O. **Sistema do BB volta a funcionar, após ficar sete horas fora do ar**. [*S.l.*: *s.n.*], 2021. https://oglobo.globo.com/economia/sistema-do-bb-volta-funcionar-apos-ficar-sete-horas-fora-do-ar-1-25174388. [Online; acessado em 24/07/2022].

GOOGLE. **LevelDB**. [*S.l.*: *s.n.*], 2021. `https://dbdb.io/db/leveldb`. [Online; acessado em 22/11/2022].

JGROUPS. **JGroups - A Toolkit for Reliable Messaging**. [*S.l.*: *s.n.*], 2002. `http://www.jgroups.org/`. [Online; acessado em 23/07/2022].

LAMPORT, Leslie. Paxos made simple. **ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)**, p. 51–58, 2001.

MICROSOFT. **Padrão embaixador**. [*S.l.*: *s.n.*], 2022. `https://docs.microsoft.com/pt-br/azure/architecture/patterns/ambassador`. [Online; acessado em 25/07/2022].

ORACLE. **Socket (Java Platform SE 8)**. [*S.l.*: *s.n.*], 2014. https://docs.oracle.com/javase/8/docs/api/java/net/Socket.html. [Online; acessado em 25/07/2022].

PEREIRA, Paola Martins; DOTTI, Fernando Luís; MEINHARDT, Cristina; MENDIZABAL, Odorico Machado. A Library for Services Transparent Replication. *In*: PROCEEDINGS of the 34th ACM/SIGAPP Symposium on Applied Computing. Limassol, Cyprus: Association for Computing Machinery, 2019. (SAC '19), p. 268–275.

SIMONE, Sergio de. **GitHub Was down Multiple Times Last February: Here's Why**. [*S.l.*: *s.n.*], 2020. `https://www.infoq.com/news/2020/03/github-february-incidents/`. [Online; acessado em 25/07/2022].

TAHTEC. **Diferença entre Unicast, multicast e broadcast**. [*S.l.*: *s.n.*], 2020. `https://tahtec.com.br/diferenca-entre-unicast-multicast-e-broadcast/`. [Online; acessado em 25/07/2022].

TANENBAUM, Andrew S. **Distributed Operating Systems**. New Jersey: Prentice Hall, 1995.

TANENBAUM, Andrew S.; STEEN, Maarten Van. **Sistemas Distribuídos: princípios e paradigmas**. 2. ed. São Paulo: Pearson Prentice Hall, 2007.

UGLIARA, Fellipe Augusto; VIEIRA, Gustavo Maciel Dias; OLIVEIRA GUIMARÃES, José de. Transparent Replication Using Metaprogramming in Cyan. *In*: PROCEEDINGS of the 21st Brazilian Symposium on Programming Languages. Fortaleza, CE, Brazil: Association for Computing Machinery, 2017. (SBLP 2017).

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

WHITE, Brian; LEPREAU, Jay; STOLLER, Leigh; RICCI, Robert; GURUPRASAD, Shashi; NEWBOLD, Mac; HIBLER, Mike; BARB, Chad; JOGLEKAR, Abhijeet. An integrated experimental environment for distributed systems and networks. **ACM SIGOPS Operating Systems Review**, ACM New York, NY, USA, v. 36, SI, p. 255–270, 2002.

APÊNDICE A – CÓDIGO DO PROJETO

O código fonte deste projeto pode ser obtido em:

- `https://github.com/carolpetrykowski/replication-library.git`

ANEXO A – ARTIGO DO PROJETO

Proposta de uma biblioteca para replicação transparente em Sistemas Distribuídos utilizando JGroups

Caroline Martins Alves[1], **Odorico Machado Mendizabal**[1],

[1] Departamento de Informática e Estatística
Universidade Federal de Santa Catarina (UFSC)
88.040-900 – Florianópolis – SC

caroline.martins@grad.ufsc.br, odorico.mendizabal@ufsc.br

Abstract. *The distributed systems development entails a series of challenges ranging from the heterogeneity of servers that host applications, through scalability and security, to the handling of failures. The present work focuses on the last aspect and, more precisely, on replication to provide fault tolerance, keeping system replicas available on different servers despite the occurrence of a limited number of failures. However, not all distributed systems implement this feature originally and, consequently, the developer is in charge of developing a replication logic suitable to the target system. This process can be expensive, time-consuming and error-prone, as it implies that the programmer has specific knowledge related to distributed systems and fault tolerance. Aiming to improve and to facilitate the use of replication in applications, the work being developed proposes the implementation of a library that offers replication in a transparent way for the programmer. Thus, systems that wish to apply this fault-tolerance strategy can easily do so, by just using our replication library. For the implementation of the library, JGroups will be used, a library for group communication.*

Resumo. *O desenvolvimento de sistemas distribuídos implica em uma série de desafios, que vão desde a heterogeneidade de servidores que hospedam aplicações, passando por escalabilidade e segurança, indo até o tratamento de falhas. O presente trabalho tem como foco o último aspecto e, mais precisamente, a replicação para oferecer tolerância a falhas, mantendo réplicas de um sistema disponíveis em diferentes servidores a despeito da ocorrência de um número limitado de falhas. Entretanto, nem todos os sistemas distribuídos implementam essa característica originalmente e, consequentemente é necessário que o desenvolvedor desenvolva a lógica de replicação especificamente para o sistema alvo. Esse processo pode ser caro, demorado e suscetível a erros, pois implica que o programador possua conhecimentos específicos relacionados a sistemas distribuídos e a tolerância a falhas. Visando melhorar e facilitar o uso de replicação em aplicações, o trabalho que está sendo desenvolvido propõe a implementação de uma biblioteca que ofereça replicação de forma transparente para o programador. Com isso, sistemas que desejem aplicar essa estratégia de tolerância a falhas podem facilmente fazê-la, apenas utilizando a nossa biblioteca. Para a implementação da biblioteca, será utilizado o JGroups, uma biblioteca para comunicação em grupo.*

1. Introdução

Com a evolução tecnológica, muitos serviços que antes só podiam ser executados fisicamente passaram a existir também no mundo digital, deixando as pessoas cada vez mais conectadas e, por vezes, dependendo dos serviços disponíveis na rede para trabalhar, para comunicar-se com família e amigos, para efetuar transações bancárias, para comércio eletrônico, entre outros. Essa mudança gerou uma grande praticidade aos usuários destes serviços, uma vez que diversas tarefas podem ser realizadas no conforto de suas casas, usando apenas um dispositivo conectado à Internet para acessar diferentes plataformas. Por outro lado, é necessário também que os provedores desses serviços garantam a sua disponibilidade e sua funcionalidade.

Quando um desses sistemas fica indisponível, acaba por gerar diversos transtornos e prejuízos para o usuário final, bem como para a empresa provedora. Para exemplificar essa situação, pode-se destacar alguns acontecimentos, como o GitHub, que, em fevereiro de 2020, esteve fora do ar diversas vezes, devido a variações inesperadas no carregamento do banco de dados e a problemas na configuração do mesmo [de Simone 2020]. Em agosto de 2021, o Banco do Brasil deixou cerca de 54 milhões de clientes sem acesso aos diversos serviços do banco [Globo 2021]. No mesmo ano, em outubro, as redes WhatsApp, Facebook, Instagram e Messenger ficaram fora do ar por quase 6 horas, causando um prejuízo de quase 6 bilhões de dólares para a Meta, empresa detentora dessas redes sociais [CNN 2021].

Nessa perspectiva, os sistemas distribuídos com ênfase em disponibilidade e em escalabilidade vêm ganhando espaço, permitindo o acesso crescente no número de usuários a um mesmo software com rapidez e com estabilidade. De acordo com Tanenbaum e Steen (2007), um sistema distribuído consiste em diversos computadores independentes, apresentados para o usuário como um único sistema coerente. Desse modo, o sistema encontra-se distribuído em diferentes computadores, aproveitando o poder de processamento de cada um destes, mas sem que o usuário perceba isso.

Entretanto, para reduzir os riscos e os impactos no caso de falhas de serviços distribuídos, espera-se que eles cumpram requisitos de sistemas confiáveis, como disponibilidade, confiabilidade, segurança e capacidade de manutenção [Verissimo and Rodrigues 2001]. Para assegurar que essas especificações sejam cumpridas, há várias técnicas de tratamento a falhas que buscam deixar os sistemas distribuídos o mais próximo possível de um sistema confiável, a exemplo da: ocultação de latência, replicação, ocultação de falhas, recuperação de falhas, entre outras.

A replicação, por sua vez, consiste em manter cópias de arquivos, de serviços ou até mesmo de sistemas inteiros, em diferentes servidores, de forma transparante para os usuários. Isso significa que, no caso de alguma réplica falhar, usuários ainda terão acesso aos dados ou à aplicação, ao conectar a uma das réplicas corretas. Implementar replicação exige coordenação entre as réplicas, uma vez que qualquer alteração feita em uma das cópias deve ser reproduzida nas demais [Charron-Bost et al. 2010]. Além disso, é necessário configurar o sistema com um número adequado de réplicas, para garantir o funcionamento correto da técnica. Portanto, a replicação não é facilmente implementada em sistemas distribuídos, necessitando que o programador implemente essa característica no sistema explicitamente. Esse processo pode ser custoso, já que requer que o desenvolvedor possua conhecimentos mais avançados em tolerância a falhas.

Visando sanar essa lacuna, este trabalho propõe a criação de uma biblioteca para replicação de serviços, de modo a prover tolerância a falhas. Consequentemente, sistemas que desejem prover replicação poderiam adicionar essa biblioteca ao seu software, sem programar do zero uma estratégia de replicação.

A seguir, na Seção 2, serão apresentados os principais conceitos e ferramentas em que esse trabalho está embasado. Na Seção 3, será abordado como pretende-se desenvolver a biblioteca, citando as tecnologias que serão utilizadas. A Seção 4 expõe alguns trabalhos que possuem correlação com a nossa proposta. Por fim, a Seção 5 apresenta as considerações finais e os próximos passos para a continuidade da pesquisa.

2. Fundamentação Teórica

Sistemas distribuídos são caracterizados por várias máquinas cooperando entre si, sem que o usuário final saiba da existência dessas diferentes máquinas ou como esse sistema opera internamente. Para isso, essa cooperação entre os diferentes computadores é essencial e, nesse sentido, tem-se que: "Um sistema distribuído é um conjunto de computadores independentes que se apresenta a seus usuários como um sistema único e coerente." [Tanenbaum and Steen 2007].

Para exemplificar esse cenário, pode-se imaginar a seguinte situação: um sistema de transações possui 3 computadores o disponibilizando de maneira íntegra, segura e confiável, em um dado momento, um desses computadores falha, porém ainda existem 2 computadores operando (Figura 1). Logo, o sistema continua disponível e o usuário continua o acessando normalmente. Isso só é possível devido à replicação, propriedade essencial desses sistemas e que será o objeto central desse trabalho. Graças a essa técnica de tolerância a falhas, não é necessário saber qual réplica está conectando ou realizar qualquer ajuste (alterar endereço, IP, porta, etc.), para continuar realizando as operações. Serviços oferecidos por bibliotecas de Comunicação em Grupo podem ser utilizados para implementar replicação, facilitando seu desenvolvimento e sua aplicação.

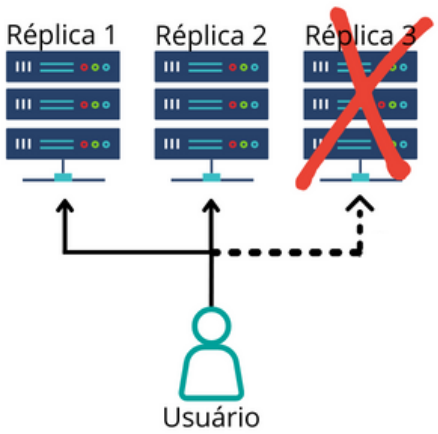


Figura 1. Abstração do funcionamento de um sistema distribuído

2.1. Comunicação em Grupo

Comunicação em grupo é definida por processos que cooperam a fim de sanar problemas de inconsistência durante a comunicação de sistemas distribuídos. Segundo

[Tanenbaum 1995], “Um grupo é um conjunto de processos cooperantes (que agem juntos) especificados pelo sistema ou por um usuário” e esses grupos podem receber e disseminar mensagens aos seus processos participantes, bem como enviar mensagens a demais grupos ou aplicações.

Assim, algumas das aplicações possíveis da comunicação em grupo são: realizar disseminação de mídias (a exemplo de um *chat*), além de tarefas mais complexas, como replicação ativa, garantia da entrega das mesmas ações em um contexto de banco de dados distribuído, entre outras.

A comunicação em grupo pode ter variações no seu tipo de comunicação, sendo *unicast*, quando as mensagens são trocadas no estilo ponto a ponto, no qual um participante de um grupo envia mensagens a um outro participante desse mesmo grupo, *multicast*, quando as mensagens são trocadas com um grupo de processos e *broadcast*, quando as mensagens são enviadas a todos os participantes do grupo. A Figura 2 apresenta o funcionamento dessas comunicações, ilustrando a comunicação um para um, um para todos e um para muitos, respectivamente.

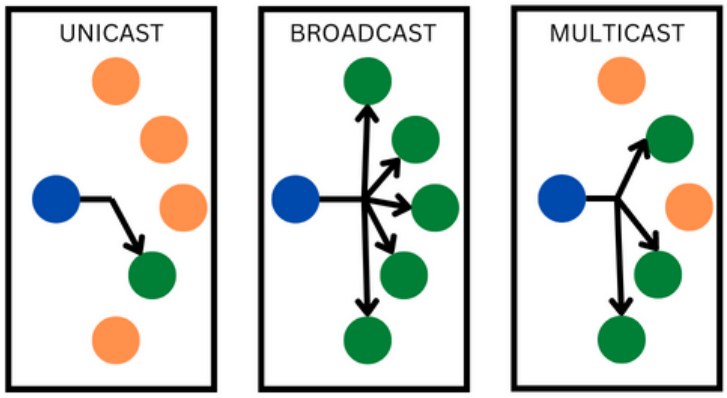


Figura 2. Tipos de comunicações no paradigma de Comunicação em Grupo (Adaptado de [TAHTEC 2020])

Em relação aos tipos de grupos, eles podem ser abertos ou fechados. Nos grupos abertos, há a interação com o ambiente que está fora do grupo, logo, aplicações externas podem enviar mensagens ao grupo e o grupo pode enviar mensagens a essas aplicações. Já nos grupos fechados, essa interação com o ambiente exterior não existe, havendo relacionamentos apenas dentro do grupo.

No quesito hierarquia, os grupos podem ser hierárquicos e não hierárquicos. No primeiro, os processos do grupo podem desempenhar diferentes papéis, como no caso de haver um nodo mestre, que controla determinadas ações e, demais nodos, que desempenham outras tarefas. No segundo, todos os participantes desempenham as mesmas funções e atividades no grupo.

Mais algumas características dos grupos são: em relação a sua dinâmica, permitindo que processos juntem-se a grupos existentes ou saiam de um grupo que fazem parte, em relação a sua confiabilidade, assegurando que a mensagem será entregue a todos os participantes ou, do contrário, será ignorada, respeitando o preceito de atomici-

dade - a mensagem chega a todos ou então a nenhum e em relação aos diferentes tipos de ordenações, que podem ser seguidos no momento da entrega dos processos, como FIFO (*First In First Out*), casual, total, entre outros.

Assim, este trabalho utilizará uma comunicação em *broadcast*, já que todos os participantes do grupo receberão as mensagens, fazendo as entregas destas com FIFO e utilizando um grupo aberto não hierárquico, pois o cliente (externo) terá interações com o grupo e todos participantes desempenharão mesma função. Este funcionamento ficará mais evidente na Seção 3, que detalha a estrutura da biblioteca.

2.2. JGroups

JGroups [Ban 2011] é uma biblioteca feita em Java que provê a troca de mensagem de maneira confiável. Com ela, é possível criar grupos com nós que podem trocar mensagens entre si. Algumas de suas principais funcionalidades são: criar e excluir grupos, entrar e sair de grupos, detecção de falhas em participantes de grupos, remoção de nós com falhas, envio e recebimento de mensagem em *unicast*, *multicast* e *broadcast*.

No contexto deste *framework*, existem dois conceitos que precisam ser entendidos, o de grupo e o de membro. Grupo é um *cluster*, processo hospedado em algum *host* e membro é um nodo que conecta-se a um grupo. Diversos nodos podem conectar-se a um *cluster* e o sistema mantém o controle dos membros que conectam ou que desconectam, notificando o *cluster* quando isso ocorre.

Já a arquitetura do JGroups, está organizada em: JChannel, *Building Blocks* e pilha de protocolos. O JChannel é usado pelos desenvolvedores para implementar aplicativos de comunicação em grupo confiáveis. Os *building blocks* ficam na camada acima do JChannel e proveem um nível maior de abstração. Já a pilha de protocolos, implementa propriedades específicas para um dado canal.

A Figura 3 representa essa estrutura, na qual o JGroups está organizado. Com isso, pode-se visualizar que um canal é conectado a uma pilha de protocolos: quando uma aplicação emite uma mensagem, o canal a transmite na pilha de protocolos, que, por sua vez, passa para o protocolo mais alto, processando a mensagem e enviando-a para o conseguinte. Logo, a mensagem é passada de protocolo a protocolo, até chegar no protocolo de transporte e ser publicada na rede.

Já o procedimento inverso ocorre para ouvir uma mensagem: o protocolo de transporte fica ouvindo a rede e, quando uma mensagem é recebida, ela é entregue para a pilha de protocolos, até chegar no canal. O canal aciona um *callback* na aplicação para entregar a mensagem. A pilha de protocolos é iniciada quando uma aplicação conecta no canal e, quando desconectar, a pilha será destruída, liberando espaço para outros recursos.

O JChannel é responsável por manusear e por controlar os grupos, permitindo desde sua criação até outras interações, que são viabilizadas pela API do JChannel. Para criar um grupo, é necessário definir um nome e todos os JChannels de mesmo nome formam um grupo, no qual seus participantes podem enviar e receber mensagens, sendo que os grupos não precisam ser criados previamente, ou seja, se um nodo liga-se a um grupo não existente, ele é automaticamente criado. Para sair do grupo, um participante pode fazer isso desconectando-se do canal e, devido sua característica de reuso, é possível que um membro que desconectou possa conectar, posteriormente, no canal. Entretanto,

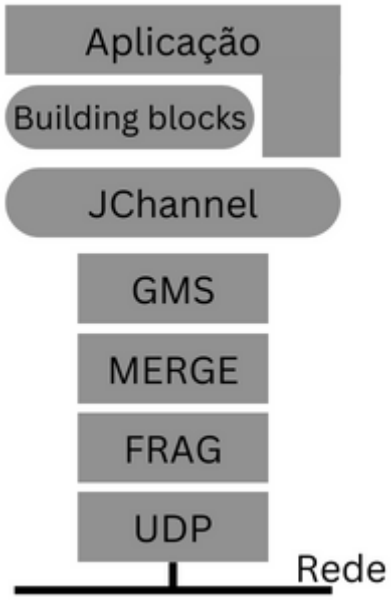


Figura 3. Arquitetura do JGroups (Adaptado de [JGroups 2002])

cada membro pode conectar apenas a um canal, não sendo possível fazer parte de mais de um canal ao mesmo tempo.

Os canais sabem quem são os membros dos grupos e, com isso, é possível gerar uma lista de endereços dos membros do canal, que é chamada de *view*. Também é viável que um processo direcione uma mensagem a um endereço da *view* ou que em *broadcast* envie uma mensagem a todos os membros, incluindo ele mesmo.

Ao invés dos canais, podem ser utilizados os *building blocks*, que ficam em uma camada acima deles e que são abstrações de nível mais alto, que concedem uma API mais sofisticada que o *JChannel*. Os blocos de construção podem criar e usar canais internos ou podem usar um canal existente para sua criação. Além disso, os *building blocks* podem oferecer acesso ao canal e, com isso, se o bloco não fornecer alguma funcionalidade, o canal pode ser acessado diretamente.

Outra propriedade muito importante da ferramenta é a sua pilha configurável de protocolos que permite a manipulação destes para atender diferentes necessidades, como particularidades de rede. Alguns dos protocolos disponíveis são: UDP (*User Datagram Protocol*), TCP (*Transmission Control Protocol*), controle de fluxo, detecção de falhas, criptografia, FIFO, além de ser possível escrever seu próprio protocolo.

Vale ainda ressaltar sua facilidade de uso e incorporação dentro de um código, sendo necessário apenas adicionar o .jar da biblioteca dentro do projeto para utilizá-lo. Ademais, a ferramenta conta com uma documentação detalhada [JGroups 2002], que permite consultar os diversos métodos da biblioteca e entender melhor seu uso e sua devida aplicação.

O JGroups é um *framework* robusto e muito bem consolidado, prova disso são as diversas empresas e soluções que o utilizam, a exemplo do JBoss, que usa o JGroups para implementação de funcionalidade de agrupamento no *JBoss J2EE Application Server*. Outro ponto é que ele incorpora de maneira excelente as premissas levantadas pela

Comunicação em Grupo, indo desde a difusão das mensagens, até níveis mais altos, como tratamento de falhas dentro do próprio grupo, e será um excelente acessório para o desenvolvimento deste projeto.

3. Desenvolvimento da Biblioteca de Replicação

A solução escolhida para implementar replicação de forma transparente foi inserir um *middleware* entre o cliente e a aplicação alvo, para que as devidas tratativas sejam feitas com o JGroups [Ban 2011] e, então, os dados sejam replicados. Logo, pretende-se desenvolver essa biblioteca, que funcionará como um interceptador, capturando pacotes enviados pelo cliente e replicando-os na aplicação alvo.

3.1. Estrutura da Ferramenta

Para que a implementação ocorra, foi necessário definir, primeiramente, sua estrutura e quais tecnologias seriam utilizadas. Será usado a ferramenta de comunicação em grupo JGroups [Ban 2011] para difusão de mensagens e para garantir todas as tratativas de confiabilidade que um sistema distribuído exige. Existem outras bibliotecas disponíveis no mercado que possuem esse mesmo propósito, a exemplo do Atomix [Foundation 2013], porém optou-se por utilizar o JGroups por ser um produto bem consolidado no mercado, sendo utilizado por servidores Tomcat JSP e JBoss J2EE [Abdellatif et al. 2004], contando com uma documentação bem detalhada, o que facilita seu uso. Optou-se pela linguagem de programação Java, uma vez que a ferramenta JGroups [Ban 2011] é escrita nela, então o uso dessa linguagem acaba por facilitar o desenvolvimento. Outro recurso escolhido foi o Java Socket [Oracle 2014], viabilizando a comunicação da biblioteca com o cliente e, posteriormente, com a aplicação alvo, que, por ser um pacote já bem difundido, é bastante prático e fácil de se utilizar.

Além disso, é essencial o uso de um padrão de projeto para a criação deste trabalho, pois esse artifício garante uma maior facilidade para manutenção e para extensão do código na adição de novas *features*. Padrões de projeto são soluções típicas para problemas comuns em projetos de softwares e, para implementação do interceptador, o padrão seguido será o embaixador [Microsoft 2022], que fornece uma maneira transparente de modificar ou de ampliar o processamento tradicional.

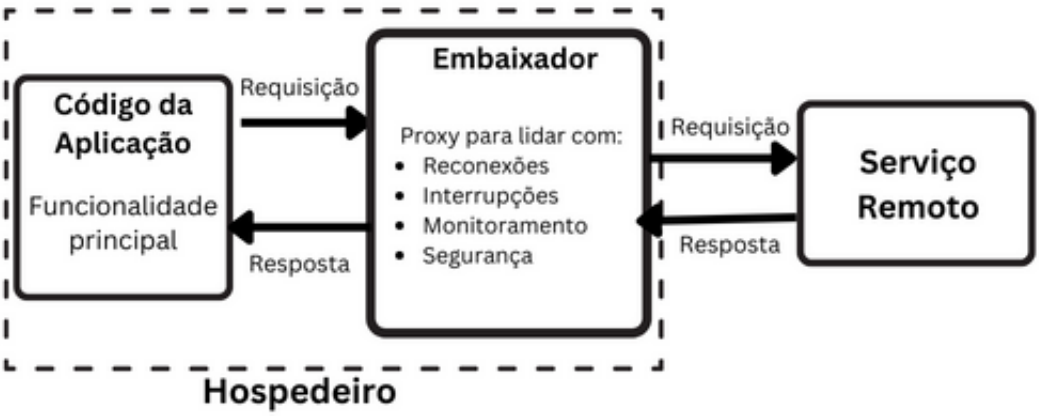


Figura 4. Exemplo do padrão de projeto embaixador (Adaptado de [Microsoft 2022])

A Figura 4 ilustra o padrão de projeto embaixador, que consiste em colocar uma estrutura (Embaixador), que será uma espécie de *proxy*, no mesmo hospedeiro que está localizado o código da aplicação, permitindo ao embaixador controlar roteamento, resiliência e recursos de segurança, dentre outras funcionalidades.

Já para a arquitetura da biblioteca que será desenvolvida, o conceito de *middleware* será adotado e, assim, um cliente enviará requisições a esta ferramenta de maneira transparente. Dentro da ferramenta será feito o uso de Comunicação em Grupo (por meio da biblioteca JGroups) e com isso será possível criar um grupo, com alguns participantes, que receberão a informação enviada pelo cliente e a entregarão nos destinos adequados, aplicando assim a replicação deste dado.

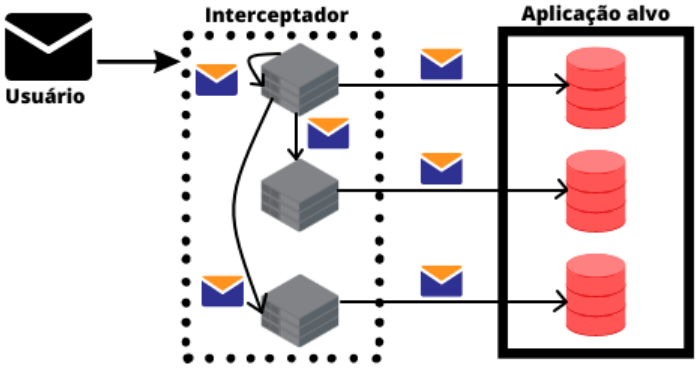


Figura 5. Funcionamento da estrutura da biblioteca

A Figura 5 ilustra o funcionamento esperado do *middleware*, em que o usuário emite uma mensagem com destino a uma aplicação, a biblioteca intercepta esse envio, capturando a mensagem e disseminando-a entre todos os processos participantes do grupo e, por fim, cada participante envia a mensagem a diferentes cópias da aplicação alvo, concretizando assim a replicação. Vale ressaltar que o interceptador será composto por n réplicas da biblioteca, que serão inicializadas e que estarão aguardando a emissão de uma requisição pelo cliente. Assume-se que até f réplicas podem falhar. Assim, o uso do padrão de projeto embaixador no desenvolvimento da biblioteca não cria um ponto único de falha, uma vez que haverá até $n - f$ réplicas ativas, sendo cada réplica responsável por uma instância replicada da aplicação alvo.

Esta arquitetura utilizada, mesmo provendo transparência de acesso entre usuário-interceptador, ainda possui algumas limitações. O cliente envia mensagens para o interceptador, sem precisar conhecê-lo, porém o interceptador precisa necessariamente conhecer os endereços da aplicação alvo para entregar essas mensagens. Além disso, até f instâncias do interceptador podem falhar e, consequentemente, estarão indisponíveis para os clientes, exigindo que o cliente descubra o endereço de outra réplica correta. Essas limitações, apesar de tornarem a solução menos transparente, não inviabilizam o seu desenvolvimento, uma vez que pode-se, por exemplo, contar com um arquivo auxiliar que mantém o controle dos endereços das réplicas e repassa ao cliente quais IPs de réplicas ativas para que as requisições sejam feitas. Alternativamente, um serviço de resolução de nomes pode ser utilizado em conjunto com a técnica.

3.2. Protótipos

Nos passos iniciais, foram desenvolvidas algumas aplicações para familiarizar-se com a ferramenta JGroups [Ban 2011], tornando possível apropriar-se de seus conceitos e métodos fundamentais. Algumas aplicações de teste foram criadas, uma delas utilizando Java e Spring Boot para a criação de *endpoints*, que serão necessários para que ocorra a comunicação do cliente com a biblioteca. Entretanto, como destacado na subseção 3.1, optou-se pelo uso de Sockets [Oracle 2014], já que a necessidade inicial é apenas fornecer um canal de comunicação do cliente com a biblioteca de replicação. Então, o uso da ferramenta Spring Boot tornou-se desnecessário, uma vez que Sockets fornecem essa mesma funcionalidade de maneira mais pontual. Essa escolha foi feita pensando em redução de custos computacionais, além de redução da complexidade da aplicação, já que o Spring Boot é uma ferramenta mais robusta e utilizada para diferentes propósitos que vão além da criação de *endpoints* e já o Sockets é mais enxuto nesse sentido.

Após o desenvolvimento de aplicações teste e familiarização com as ferramentas, foi possível dar início ao desenvolvimento da biblioteca. O que se tem, até agora, é uma aplicação que está ativa com um grupo de três participantes (este valor foi escolhido arbitrariamente e pode ser alterado posteriormente) aguardando receber uma solicitação de um cliente via Socket. Em um segundo momento, será desenvolvido o envio dessa solicitação do cliente a aplicação alvo por cada um dos três participantes do grupo.

A aplicação alvo escolhida para ser estudo de caso dessa solução é o banco de dados RocksDB [Facebook 2020], um banco em memória que não implementa replicação. Inicialmente, usaremos este banco para os primeiros testes e validações, mas a ideia do trabalho é chegar em um nível de generalização, em que a solução funcione independentemente de qual seja a aplicação alvo.

3.3. Teste e avaliação

Após o desenvolvimento da biblioteca estar completo, serão necessárias algumas verificações para confirmar se a ferramenta, além de resolver o problema inicial, o faz de maneira eficaz, sem gerar custos de desempenho demasiados que invalidem a execução da aplicação, por exemplo. É esperado que o uso da solução gere algum gasto computacional a mais, como o aumento da latência e vazão, uma vez que, ao invés do cliente entregar a mensagem diretamente ao destino, agora ela passa primeiramente pelo interceptador.

Para os testes, será utilizada a plataforma Emulab [White et al. 2002]. O Emulab é um ambiente de *testbed* que fornece o suporte necessário para alocar nodos e para configurar características de rede em Sistemas Distribuídos em ambiente controlado. Dentre os testes, algumas comparações devem ser feitas, a fim de validar a qualidade da solução, como, verificar o desempenho obtido quando uma aplicação utiliza a biblioteca e o desempenho quando a aplicação implementa replicação sem o suporte da biblioteca. Pretende-se instrumentar os serviços e os clientes para coleta de vazão de requisições processadas, bem como latência na resposta de requisições. Considerando execuções sucessivas, com acréscimo na carga de trabalho, deve-se produzir um gráfico que ilustre os valores de vazão e de latência, permitindo observar os pontos de saturação (*i.e.*, quando a vazão estabiliza e a latência passa a aumentar significativamente). A comparação de desempenho seguirá uma abordagem similar à proposta em [Pereira et al. 2018]. Não será estipulado um critério de aceitação, uma vez que o objetivo dos testes será avaliar o custo ocasionado

pela incorporação da biblioteca para replicação em uma aplicação.

4. Trabalhos relacionados

Nesta seção serão apresentados alguns trabalhos que possuem correlação com este projeto e propõem fornecer replicação de forma transparente, apontando para formas de implementar essa técnica de tolerância a falhas.

Em [Pereira et al. 2019], é apresentada uma estratégia para replicar trechos de código de um sistema, utilizando a abordagem de Replicação de Máquina de Estados (RME), por meio do protocolo de consenso Paxos [Lamport 2001]. Assim, para que o programador determine que certo método deve ser replicado, é necessário que utilize a anotação *@SmrMethod* (com seus devidos parâmetros), logo acima da assinatura do método. A ferramenta identificará, com base nas anotações de código, quais métodos devem ser replicados. Desse modo, o serviço de replicação oferecido pela biblioteca aumenta a disponibilidade ao custo de uma perda de desempenho. Além disso, foi observada uma perda de vazão ao utilizá-la, como já era esperado. Houve também um aumento da latência ocasionado pelo uso do protocolo Paxos [Lamport 2001] e não da biblioteca em si. Essa ferramenta possui um intuito muito parecido com o proposto neste trabalho, uma vez que os dois tratam de replicação em sistemas distribuídos de maneira transparente. Entretanto, a biblioteca de replicação de serviços exige que o programador anote os métodos que devem ser replicados, necessitando de uma intervenção do desenvolvedor e uma alteração da aplicação alvo para utilizá-la. Diferentemente, a nossa proposta visa integrar um serviço de replicação na forma de um interceptador, capturando o tráfego de requisições entre cliente e aplicação alvo. Espera-se que essa abordagem evite modificações na aplicação a ser replicada.

A proposta feita por [Ugliara et al. 2017] apresenta como usar a estrutura de meta programação da linguagem Cyan, em prol de desenvolver softwares replicados e tolerantes a falha por meio de anotações simples, sem que o programador precise saber detalhes profundos de replicação. Isso pode ser feito por meio de meta objetos, que podem ser anexados a métodos modificadores. O artigo mostra como fazer essa implementação, gerando e validando o código de integração que usa a estrutura de Treplica. Os autores também apresentam como a ferramenta faz verificações em relação a determinismo, na qual o desenvolvedor é alertado caso exista alguma operação inconsistente ou método modificador. Como trabalhos futuros, os autores pretendem criar um mecanismo capaz de substituir essas operações não determinísticas por operações determinísticas equivalentes. Assim como o trabalho apresentado em [Pereira et al. 2019], os autores propõem uma maneira facilitada de implementar replicação em sistemas. Entretanto, o artigo ainda exige que uma linguagem de programação (Cyan) específica seja utilizada para isso, enquanto este projeto almeja que a replicação seja possível de ser utilizada em diferentes contextos e sem a exigência de que seja feita em uma linguagem de programação específica.

5. Considerações Finais

Ao fim deste trabalho, espera-se obter uma ferramenta que atuará como um interceptador, replicando mensagens que são enviadas de um cliente com destino a alguma aplicação, utilizando inicialmente o RocksDB como caso de teste desta solução e, posteriormente, podendo expandi-la para demais aplicações.

Para validar o sucesso dessa biblioteca, serão realizados testes em ambiente de simulação de Sistemas Distribuídos, o Emulab, avaliando os custos agregados a essa ferramenta e quais os impactos causados por ela. Com isso, espera-se produzir avaliações (em formato de gráfico), que permitam visualizar e concluir qual o custo agregado causado pela solução em termos de vazão e de latência.

Atualmente, o que já foi desenvolvido foram pesquisas em torno dos temas base deste projeto, como Comunicação em Grupo e a própria ferramenta JGroups, além da avaliação do estado da arte nessa área (vide Seção 4). Assim, para finalização deste trabalho, o protótipo já desenvolvido precisa ser aprimorado e ter o seu desempenho avaliado, de modo a permitir ajustes e a finalização de sua implementação na forma de uma biblioteca de replicação.

Agradecimentos

O presente trabalho foi realizado com o apoio do Fundo Nacional de Desenvolvimento da Educação - FNDE e do PET Computação - UFSC.

Referências

Abdellatif, T., Cecchet, E., and Lachaize, R. (2004). Evaluation of a group communication middleware for clustered J2EE application servers. In *OTM Confederated International Conferences On the Move to Meaningful Internet Systems*, pages 1571–1589. Springer.

Ban, B. (2011). Reliable Multicasting with the JGroups Toolkit. http://www.jgroups.org/manual/html_single/. [Online; acessado em 15/12/2021].

Charron-Bost, B., Pedone, F., and Schiper, A. (2010). *Replication: Theory and Practice*, volume 5959. Springer.

CNN (2021). Mark Zuckerberg perde quase US$ 6 bi em dia de falhas e denúncias ao Facebook. https://www.cnnbrasil.com.br/business/mark-zuckerberg-perde-quase-us-6-bi-em-dia-de-falhas-e-denuncias-ao-facebook/. [Online; acessado em 04/03/2022].

de Simone, S. (2020). GitHub Was down Multiple Times Last February: Here's Why. https://www.infoq.com/news/2020/03/github-february-incidents. [Online; acessado em 25/07/2022].

Facebook (2020). RocksDB. http://rocksdb.org/. [Online; acessado em 25/07/2022].

Foundation, O. N. (2013). Atomix Framework. https://atomix.io/. [Online; acessado em 14/11/2022].

Globo, O. (2021). Sistema do BB volta a funcionar, após ficar sete horas fora do ar. https://oglobo.globo.com/economia/sistema-do-bb-volta-funcionar-apos-ficar-sete-horas-fora-do-ar-1-25174388. [Online; acessado em 24/07/2022].

JGroups (2002). JGroups - A Toolkit for Reliable Messaging. http://www.jgroups.org/. [Online; acessado em 23/07/2022].

Lamport, L. (2001). Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58.

Microsoft (2022). Padrão embaixador. https://docs.microsoft.com/pt-br/azure/architecture/patterns/ambassador. [Online; acessado em 25/07/2022].

Oracle (2014). Socket (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/net/Socket.html. [Online; acessado em 25/07/2022].

Pereira, P., Müller, R. H., and Mendizabal, O. M. (2018). Avaliação de desempenho de bibliotecas java para o protocolo paxos. In *Anais da XVIII Escola Regional de Alto Desempenho do Estado do Rio Grande do Sul*, Porto Alegre, RS, Brasil. SBC.

Pereira, P. M., Dotti, F. L., Meinhardt, C., and Mendizabal, O. M. (2019). A library for services transparent replication. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC '19, page 268–275, New York, NY, USA. Association for Computing Machinery.

TAHTEC (2020). Diferença entre Unicast, multicast e broadcast. https://tahtec.com.br/diferenca-entre-unicast-multicast-e-broadcast/. [Online; acessado em 25/07/2022].

Tanenbaum, A. S. (1995). *Distributed Operating Systems*. Prentice Hall, New Jersey.

Tanenbaum, A. S. and Steen, M. V. (2007). *Sistemas Distribuídos: princípios e paradigmas*. Pearson Prentice Hall, São Paulo, 2. ed. edition.

Ugliara, F. A., Vieira, G. M. D., and de Oliveira Guimarães, J. (2017). Transparent replication using metaprogramming in cyan. In *Proceedings of the 21st Brazilian Symposium on Programming Languages*, SBLP 2017, New York, NY, USA. Association for Computing Machinery.

Verissimo, P. and Rodrigues, L. (2001). Distributed systems for system architects.

White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A. (2002). An integrated experimental environment for distributed systems and networks. *ACM SIGOPS Operating Systems Review*, 36(SI):255–270.