UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
SISTEMAS DE INFORMAÇÃO

Rafaela Tillmann

Middle-earth: Implementação de um Middleware para Replicação em Sistemas Distribuídos

Florianópolis
2025

Rafaela Tillmann

Middle-earth: Implementação de um Middleware para Replicação em Sistemas Distribuídos

Trabalho de Conclusão de Curso submetido ao Curso de Graduação em Sistemas de Informação do Centro Tecnológico da Universidade Federal de Santa Catarina como requisito para obtenção do título de Bacharel em Sistemas de Informação.

Orientador: Prof. Odorico Machado Mendizabal, Dr.

Florianópolis
2025

Ficha catalográfica gerada por meio de sistema automatizado gerenciado pela BU/UFSC.
Dados inseridos pelo próprio autor.

Tillmann, Rafaela
Middle-earth: Implementação de um Middleware para Replicação em Sistemas Distribuídos / Rafaela Tillmann ; orientador, Odorico Machado Mendizabal, 2025.
69 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Sistemas de Informação, Florianópolis, 2025.

Inclui referências.

1. Sistemas de Informação. 2. Middleware. 3. Replicação. 4. Sistemas Distribuídos. 5. Log distribuído. I. Mendizabal, Odorico Machado. II. Universidade Federal de Santa Catarina. Graduação em Sistemas de Informação. III. Título.

Rafaela Tillmann

Middle-earth: Implementação de um Middleware para Replicação em Sistemas Distribuídos

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de Bacharel em Sistemas de Informação e aprovado em sua forma final pelo curso de Graduação em Sistemas de Informação.

Florianópolis, 2025.

Álvaro Junio Pereira Franco, Dr.
Coordenador do Curso

Banca Examinadora:

Prof. Odorico Machado Mendizabal, Dr.
Orientador
Universidade Federal de Santa Catarina

Prof. Patricia Della Mea Plentz, Dra.
Avaliador
Universidade Federal de Santa Catarina

Prof. Eduardo Camilo Inacio, Dr.
Avaliador
Universidade Federal de Santa Catarina

Thinking doesn't guarantee that we won't make mistakes. But not thinking guarantees that we will – Leslie Lamport

RESUMO

Na sociedade pós-moderna o imediatismo requerido pela humanidade gera uma urgência por um fluxo contínuo de informações, logo, a indisponibilidade de um serviço pode acarretar vários prejuízos. Para assegurar a disponibilidade contínua dos serviços, é possível replicá-los em múltiplos servidores, seguindo abordagens como a replicação passiva (ou *primary-backup*) ou a replicação ativa (ou Replicação de Máquina de Estados). Na replicação passiva, apenas uma réplica é responsável pelo processamento das requisições, enquanto as outras atuam como *backups*. Já na replicação ativa, prevê-se que todas as réplicas irão atuar no processamento das solicitações. Para tal, é necessário que todas as réplicas executem a mesma sequência de operações determinísticas para, assim, evoluírem por estados idênticos. Entretanto, devido aos desafios associados à implementação dessas duas estratégias, este trabalho visa o desenvolvimento do Middle-earth, um *middleware* para replicação, que atue como um serviço intermediário entre os clientes e as réplicas de uma determinada aplicação, assegurando consistência forte entre as réplicas e proporcionando transparência no processo de replicação. O Middle-earth é constituído por dois serviços: um *log* distribuído e um serviço denominado *logger*. O *log* distribuído é responsável pela durabilidade e disponibilidade dos dados, enquanto o *logger* é responsável por efetivamente interceptar as requisições e coordená-las para as réplicas. Com o objetivo de identificar o custo associado ao Middle-earth, foi executada uma avaliação experimental. Nesta, o desempenho do *middleware* apresentou-se inferior ao de uma estratégia sem replicação. Contudo, esse é um custo inerente à replicação e ao processo de interceptação. A avaliação também permitiu constatar que o *middleware* demonstrou capacidade para suportar acréscimos de servidores em sua carga de trabalho, evidenciando seu potencial de escalabilidade.

Palavras-chave: *Middleware*. Replicação. Sistemas Distribuídos. *Log* distribuído.

ABSTRACT

In post-modern society, the immediacy required by humanity generates an urgency for a continuous flow of information, thus, the unavailability of a service can lead to several losses. To ensure the continuous availability of services, it is possible to replicate them across multiple servers following approaches such as passive replication (or primary-backup) or active replication (or State Machine Replication). In passive replication, only one replica is responsible for processing requests, while the others act as backups. In active replication, it is expected that all replicas will act in processing requests. For this purpose, it is necessary that all replicas execute the same sequence of deterministic operations to evolve through identical states. However, due to the challenges associated with implementing these two strategies, this work aims at the development of Middle-earth, a replication middleware, which acts as an intermediate service between clients and the replicas of a given application, ensuring strong consistency among the replicas and providing transparency in the replication process. Middle-earth consists of two services: a distributed log and a service called logger. The distributed log is responsible for data durability and availability, while the logger is responsible for effectively intercepting requests and coordinating them to the replicas. To identify the cost associated with Middle-earth, an experimental evaluation was conducted. In this evaluation, the middleware's performance was inferior to that of a non-replicated strategy. However, this is a cost inherent to replication and the interception process. The evaluation also demonstrated that the middleware has the capacity to support the addition of servers to its workload, evidencing its potential for scalability.

Keywords: Middleware. Replication. Distributed Systems. Distributed log.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

LISTA DE ABREVIATURAS E SIGLAS

2PC	*Two-phase commit*
API	*Application Programming Interface*
AWS	Amazon Web Services
CAS	*compare-and-swap*
EC2	*Elastic Compute Cloud*
ERAD	Escola Regional de Alto Desempenho da Região Sul
JDBC	*Java Database Connectivity*
kB	*kilobyte*
LAC	*LastAddConfirmed*
LAP	*LastAddPushed*
ms	milissegundos
POC	*Proof of Concept*
RME	Replicação de Máquinas de Estados
SaaS	*Software as a Service*
SGBD	Sistema de Gerenciamento de Banco de Dados
SMR	*State Machine Replication*
SPOF	*Single Point of Failure*
WAL	*Write-ahead logging*

SUMÁRIO

1 INTRODUÇÃO

A computação em nuvem emergiu a partir da expansão da Internet e, atualmente, é um paradigma já consolidado que oferta recursos virtualizados através de uma infraestrutura compartilhada entre vários clientes a partir do método *pay-per-use*. Este cenário possibilitou que emergissem soluções em variadas áreas, visto que as organizações não tem mais a necessidade de dedicar-se a sua infraestrutura, para que o foco possa ser apenas no negócio e no desenvolvimento de suas aplicações.

Essa alteração de paradigma transformou o fluxo de informações na moeda das organizações, em que a indisponibilidade de um serviço pode acarretar em vários prejuízos. Como foi para a Amazon em 2021, quando a indisponibilidade de apenas 59 minutos acarretou em uma perda de 34 milhões de dólares em vendas (The Independent, 2021). Outro caso foi a indisponibilidade da Amazon Web Services (AWS) em 2025, infraestrutura que sustenta inúmeros sites e plataformas na Internet e, portanto, afetou milhares de serviços, como Canva, Snapchat, Duolingo e Zoom (BBC News Brasil, 2025).

Portanto, para assegurar a disponibilidade dos serviços, é importante implementar estratégias de tolerância a falhas. Dentre as opções, a replicação de serviços é a comumente aplicada, como a replicação passiva (*primary-backup*) (BUDHIRAJA et al., 1993) e a replicação ativa (Replicação de Máquina de Estados) (SCHNEIDER, 1990; SCHNEIDER, 1993; LAMPORT, 1978). É a partir dessas estratégias de replicação que sistemas de alta disponibilidade são construídos (CORBETT et al., 2013; VERBITSKI et al., 2017; JUNQUEIRA; REED, 2013).

Na replicação passiva um único servidor, denominado primário, é responsável por processar as requisições dos clientes e, após a execução da requisição, replicar o estado atualizado para os servidores secundários (*backups*). Apesar de assegurar confiabilidade, essa técnica gera desequilibro de carga, visto que a réplica primária atua no processamento de todas as solicitações dos clientes, enquanto as outras réplicas atuam apenas na aplicação das atualizações. Outro ponto associado é a consistência eventual entre as réplicas, visto que há um tempo até a atualização integral do estado entre todas elas. Além da possibilidade de ocasionar a indisponibilidade temporária do serviço no período de substituição do primário, caso este seja acometido por uma falha (BUDHIRAJA et al., 1993).

Na Replicação de Máquina de Estados é previsto que todas as réplicas iniciem um mesmo estado e executem uma sequência idêntica de operações determinísticas, avançando, assim, por estados idênticos. Nesta abordagem, há necessidade de estabelecer um acordo quanto a ordenação das requisições entre as réplicas, este assegurado por protocolos de consenso (LAMPORT, 1998; ONGARO; OUSTERHOUT, 2014) ou de difusão atômica (CRISTIAN et al., 1995). Os protocolos são estruturados em nível de aplicação, fato que eleva complexidade de

implementação, visto que o projetista deve possuir conhecimento desses protocolos e outras técnicas de coordenação entre as réplicas, além da impossibilidade de aplicar a solução de replicação para outros serviços, de forma genérica (SCHNEIDER, 1990; SCHNEIDER, 1993; LAMPORT, 1978). Além disto, a replicação ativa é restrita apenas a operações determinísticas, fato que limita a aplicação a execuções sequenciais e, consequentemente, a implicações no desempenho.

Este trabalho visa contornar os obstáculos inerentes à implementação dessas estratégias. Dentre estes, destaca-se a complexidade de implantação, que exige conhecimentos específicos sobre tolerância a falhas, e a inviabilidade de reutilização do modelo de replicação em aplicações distintas.

Com isso, a monografia propõe o desenvolvimento do Middle-earth, um serviço de replicação como *middleware*, alocado entre os clientes e as réplicas de uma determinada aplicação para atuar na coordenação da replicação. A partir do deslocamento da estratégia de replicação das réplicas para um serviço externo, o projeto almeja proporcionar alta modularidade, independência, escalabilidade e, principalmente, otimizações no desenvolvimento, já que não há necessidade de conhecimentos específicos de estratégias de replicação para implantar o Middle-earth entre um serviço.

A proposta deste trabalho foi publicada na XXIV Escola Regional de Alto Desempenho da Região Sul (ERAD). Essa publicação consiste na análise preliminar efetuada no período de elaboração do trabalho e demonstra a trajetória do projeto quando comparada com a presente monografia.

1.1 OBJETIVOS

1.1.1 Objetivo Geral

O desenvolvimento do Middle-earth, um *middleware* para replicação em sistemas distribuídos. O *middleware* será responsável por atuar na intercepção das requisições dos clientes de uma determinada aplicação e, apenas após assegurar a ordenação e durabilidade, transmitilas para as réplicas da aplicação.

1.1.2 Objetivos Específicos

1. Projetar a arquitetura do Middle-earth;
2. Implementar uma POC (*Proof of Concept*) do Middle-earth a partir da arquitetura idealizada;

3. Executar uma avaliação experimental do Middle-earth com objetivo de identificar o custo associado ao *middleware*.

1.2 ESTRUTURA DO TRABALHO

O restante deste trabalho está organizado da seguinte forma: O Capítulo 2 discorre quanto a fundamentação teórica, discutindo estratégias de replicação e dois serviços aplicados no desenvolvimento do Middle-earth. O Capítulo 3 discute os trabalhos correlatos. O Capítulo 4 apresenta o Middle-earth, sua arquitetura e a implementação de uma *Proof of Concept* (POC) do *middleware*. No Capítulo 5 são apresentados os resultados da avaliação experimental do *middleware* e, por fim, o Capítulo 6 discorre as considerações finais do trabalho e possíveis trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo é apresentado a fundamentação, que engloba replicação, mais especificamente, duas estratégias centrais de replicação e dois serviços adotados no desenvolvimento da uma POC (*Proof of Concept*) do projeto: o Apache ZooKeeper, um serviço de coordenação distribuído, e o Apache BookKeeper, um serviço de *logs* distribuídos.

2.1 REPLICAÇÃO

Visto a necessidade da disponibilidade contínua dos serviços, estratégias de replicação são comumente aplicadas para prover tolerância a falhas, assegurando a execução do serviço mesmo na ocorrência de falhas em um número limitado de réplicas. Dentre as várias estratégias de replicação, há destaque para a replicação passiva (ou *primary-backup*) e a replicação ativa (ou Replicação de Máquinas de Estados).

2.1.1 Replicação de Máquinas de Estado

Na Replicação de Máquinas de Estado (RME ou SMR, do inglês, *State Machine Replication*) ou simplesmente replicação ativa é previsto que um conjunto de réplicas inicie em um mesmo estado e execute uma sequência ordenada de operações determinísticas para, assim, avançar por estados idênticos (SCHNEIDER, 1990; SCHNEIDER, 1993; LAMPORT, 1978). Portanto, essa estratégia requer que o cliente envie as solicitações para todas as réplicas e, após a entrega da requisição, há a necessidade de estabelecer um acordo quanto a ordenação correta entre as réplicas. Nessa perspectiva, a implementação da replicação ativa deve assegurar três requisitos, coordenação, acordo e ordenação:

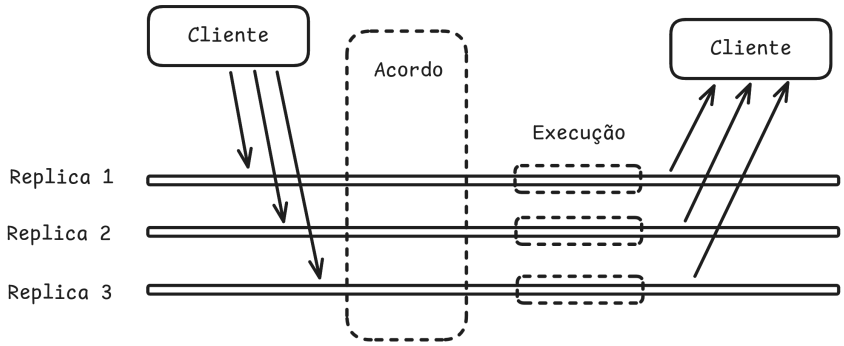
- A coordenação assegura que todas as réplicas irão receber e processar as mesmas requisições;
- O acordo define que todas as réplicas não falhas irão receber todas as solicitações;
- A ordenação assegura que todas as réplicas não falhas irão executar as requisições numa mesma ordem.

A ordenação é assegurada, normalmente, por protocolos de consenso, como o Paxos (LAMPORT, 1998) e o Raft (ONGARO; OUSTERHOUT, 2014), ou por algoritmos de difusão atômica (CRISTIAN et al., 1995). Para fins de ilustração, são utilizado protocolos de consenso

para exemplificar a dinâmica entre as réplicas. Os protocolos de consenso têm suas particularidades, todavia, a arquitetura tipicamente aplicada na Replicação de Máquina de Estados prevê que as réplicas irão possuir papéis distintos, dos quais uma réplica irá atuar como líder e será responsável por propor operações para os outros nós. Após a confirmação por um quórum de réplicas quanto a proposta, um quórum é definido como $(n/2) + 1$, no qual n é o número de réplicas operantes, elas irão executar a operação e enviar o resultado da requisição para o cliente, como exemplificado na Figura 1.

Entretanto, tais protocolos são estruturados em nível de aplicação, fato que eleva a complexidade associada à implementação de aplicações tolerantes a falhas, uma vez que o projetista deve possuir conhecimento dos protocolos e de outras técnicas de coordenação entre as réplicas, além da impossibilidade de aplicar a solução de replicação a outros serviços. Outro fator associado a replicação ativa é a latência inerente a estratégia, já que as réplicas devem estabelecer uma comunicação entre elas a fim de determinar a ordenação das solicitações.

Figura 1 – Replicação ativa



Fonte: Elaborada pela autora

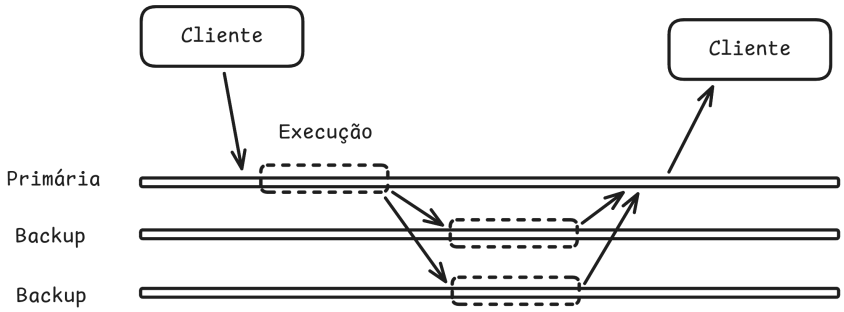
Além disso, apesar da estratégia assegurar consistência forte entre as réplicas, ela é restrita apenas a operações determinísticas, fato que limita a aplicação a execuções sequenciais, já que execuções paralelas são, normalmente, não-determinísticas. Apesar da existência de propostas de Replicação de Máquinas de Estado Paralela (ALCHIERI et al., 2018; MARANDI; BEZERRA; PEDONE, 2014; MENDIZABAL; DOTTI; PEDONE, 2017), ao avaliar a natureza da requisição e sua dependência com as demais, há algumas restrições, como a necessidade de conhecimento da semântica da aplicação para determinar se a execução será concorrente ou, obrigatoriamente, sequencial.

2.1.2 Replicação passiva

Na replicação passiva (ou *primary-backup*), diferentemente da replicação ativa, apenas uma réplica é responsável pelo processamento das requisições dos clientes. Após essa réplica

primária processar a solicitação do cliente, ela deve transmitir o estado final para as demais réplicas, que irão atuar apenas como *backups*, através de um protocolo *multicast*, conforme ilustrado na Figura 2 (BUDHIRAJA et al., 1993). Caso seja identificada uma falha na réplica primária, um *backup* deve assumir a posição primária e o fluxo de processamento prossegue como outrora.

Figura 2 – Replicação passiva



Fonte: Elaborada pela autora

Nesta estratégia, apenas o servidor primário atua como uma máquina de estado, logo, operações não determinísticas são permitidas, visto que os *backups* irão apenas aplicar as atualizações e não processá-las. Porém, esse cenário gera um desequilíbrio de carga, pois enquanto a réplica primária atua no processamento de todas as solicitações dos clientes, as outras réplicas irão apenas atuar na aplicação das atualizações, resultando em um único ponto de falha dentro da arquitetura.

A replicação passiva também pode elevar a latência das requisições, visto que após o processamento da requisição o servidor primário deve replicar o estado para um quórum de servidores *backups* antes de efetivamente responder ao cliente. Apesar da possibilidade de responder o cliente logo após o processamento da requisição, essa ação não assegura consistência forte entre as réplicas, além da probabilidade do servidor ser substituído por uma réplica em um estado anterior, caso este seja acometido por uma falha. Outra questão neste cenário de falha, é a possibilidade de ocasionar a indisponibilidade temporária do serviço no período de substituição do primário

2.2 APACHE ZOOKEEPER

O ZooKeeper[1] é um serviço de coordenação distribuído projetado para mitigar a dificuldade inerente à coordenação de sistemas distribuídos. “ZooKeeper” é uma metáfora na qual os sistemas distribuídos são comparados a zoológicos, devido a complexidade inerente a

[1] https://zookeeper.apache.org

estes que o ZooKeeper visa contornar. O serviço visa disponibilizar uma solução escalável e reutilizável para possibilitar que haja concentração por parte dos desenvolvedores apenas na lógica das aplicações no lugar de estratégias de coordenação. Desenvolvido originalmente no Yahoo! Research, atualmente o ZooKeeper é um projeto de código aberto hospedado pela Apache Software Foundation (JUNQUEIRA; REED, 2013; HALOI, 2015; The Apache Software Foundation, 2025b).

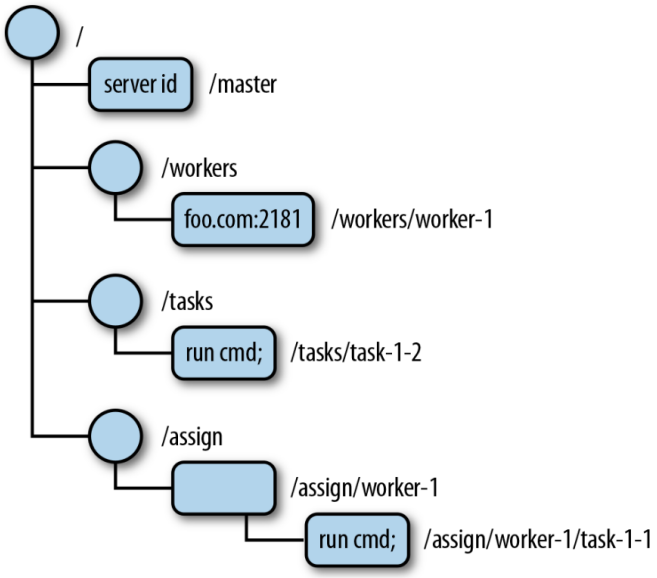
O serviço expõe uma API (*Application Programming Interface*) que permite a implementação de tarefas de coordenação, como *leader election* e *locks* distribuídos. Todavia, o ZooKeeper não se concentra apenas na aplicação de primitivas de sincronização, é possível implementar a coordenação de quaisquer metadados entre servidores.

O ZooKeeper não expõe primitivas de coordenação diretamente, por exemplo, *lock* e *unlock* para *locks* distribuídos. No lugar, ele disponibiliza uma API *file system-like* composta por um pequeno conjunto de funções responsáveis por manipular nós denominados *znodes*, que são organizados hierarquicamente como uma árvore, tal qual um sistema de arquivos. A organização hierárquica permite que uma aplicação implemente suas próprias primitivas de coordenação, que são denotadas de *recipes* dentro do ZooKeeper. A Figura 3 ilustra essa organização hierárquica de um *znode*. Nela, o nó raiz ("/") contém outros nós, e essa sucessão prossegue até os nós folha, que são efetivamente os dados. A própria estruturação do *znode* é capaz de transmitir informações, por exemplo, a ausência de um *znode*, dentro da arquitetura *master-worker* ilustrada, significa que ainda nenhum mestre foi eleito. A Figura 3 incluí outros *znodes* para essa configuração *master-worker*:

- O *znode* $/workers$ é o *znode* pai de todos os *znodes* que representam um trabalhador disponível. A Figura demonstra que o trabalhador "$foo.com$: 2181" está disponível. Se um trabalhador ficar indisponível, o *znode* deve ser removido de $/workers$.
- O *znode* $/tasks$ é o pai de todas as tarefas pendentes de execução por parte dos trabalhadores.
- O *znode* $/assign$ é o *znode* pai de todas as tarefas com um trabalhador atribuído. Quando um mestre atribui uma tarefa a um trabalhador, ele adiciona um *znode* em $/assign$.

Caso um *znode* contenha dados, estes são incluídos como um *array* de *bytes*, em que o formato exato é específico para cada aplicação. Os dados armazenados nos *znodes* irão consistir exclusivamente de metadados, pois o ZooKeeper não é destinado ao armazenamento em massa (*bulk storage*). Além disso, o ZooKeeper não permite escrita ou leitura parcial do conteúdo de um *znode*, o conteúdo é sempre substituído ou lido inteiramente. A API expõe as seguintes operações para a manipulação dos *znodes*:

Figura 3 – Estrutura de um *znode*



Fonte: Junqueira e Reed (2013)

- *create /path data*: Cria um *znode* no *path* especificado contendo dados;
- *delete /path*: Excluí o *path* indicado;
- *exists /path*: Verifica se o *path* existe;
- *setData /path data*: Registra o dado informado no *path* indicado;
- *getData /path*: Captura o dado no *path* indicado;
- *getChildren /path*: Devolve uma lista de *znodes* contidos no *path* informado.

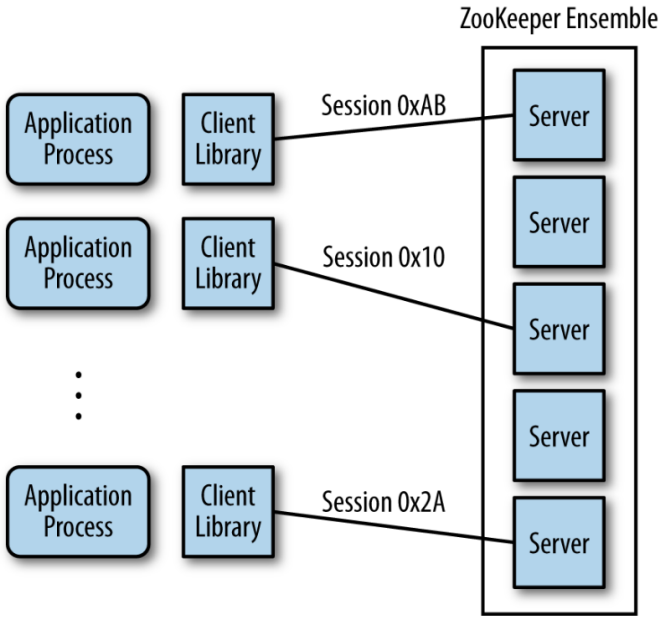
Um *znode* possui dois modos distintos: persistente (*persistent*) ou efêmero (*ephemeral*). Um *znode* persistente só pode ser excluído a partir de uma solicitação direta de exclusão, já um *znode* efêmero, em contraste, é excluído sempre que a sessão com o ZooKeeper é encerrada. Um *znode* pode, ainda, ser designado como sequencial (*sequential*). Um número inteiro único e sempre crescente é atribuído e anexado ao *path* do *znode* ($/tasks/task-1$). *Znodes* sequenciais proveem facilidade na nomeação de *znodes*, além de permitir a visualização da ordem destes. Há quatro combinações possíveis de modos: *persistent*, *ephemeral*, *persistent_sequential* e *ephemeral_sequential*.

2.2.1 Arquitetura

A arquitetura do ZooKeeper é ilustrada na Figura 4, na qual os clientes deverão solicitar as operações através de uma biblioteca (*client library*) inerente a aplicação. O ZooKeeper

client, a partir dos métodos discutidos, estabelece a interação com os servidores do ZooKeeper. Contudo, antes de solicitar qualquer operação para o ZooKeeper o cliente deve estabelecer uma sessão com o serviço. O cliente, a princípio, conecta-se apenas a um único servidor, porém caso seja identificado um *timeout* na conexão, o ZooKeeper *client* pode transmitir a sessão para outro servidor (JUNQUEIRA; REED, 2013; The Apache Software Foundation, 2025b).

Figura 4 – Arquitetura do ZooKeeper



Fonte: Junqueira e Reed (2013)

O ZooKeeper opera em dois modos distintos, *standalone* e quórum. No modo *standalone* um único servidor é usado e o estado do ZooKeeper não é replicado. No modo quórum, um grupo de servidores, denominados "ZooKeeper *ensemble*", replica o estado entre as várias instâncias de servidores.

Um quórum é definido como o número mínimo de servidores ativos para que ZooKeeper opere e, também, o número mínimo de servidores que deverão confirmar a operação antes de prosseguir com a devolutiva para o cliente. Assim, um quórum é determinado por $(n/2) + 1$, em que n é o número de servidores no *ensemble*. Ao aplicar este esquema, o serviço é capaz de tolerar até f falhas, em que f é o número de servidores excedentes ao quórum. Um ZooKeeper *ensemble* de, por exemplo, 5 servidores possui um quórum de 3 servidores e pode tolerar até 2 falhas.

2.3 APACHE BOOKKEEPER

O Apache BookKeeper[2] é um serviço de *logs* distribuídos que proporciona durabilidade, replicação e consistência forte, utilizado por várias aplicações no desenvolvimento de *write-ahead logging* (WAL) (GRACIA-TINEDO et al., 2023; WHITE, 2012) ou para persistência de informações (SHARMA; ATYAB, 2022). Originalmente desenvolvimento pela Yahoo! como um sub projeto do Apache ZooKeeper, atualmente está oficialmente incorporado à Apache Software Foundation como um projeto independente (JUNQUEIRA; KELLY; REED, 2013; The Apache Software Foundation, 2025a).

Inúmeras aplicações têm a necessidade de assegurar a durabilidade das suas atualizações e uma técnica comum para disponibilizar durabilidade é a implementação de *write-ahead logging* (WAL). Contudo, se o *log* residir apenas localmente, essa ação vincula a recuperabilidade dos dados à recuperabilidade do servidor. Com isso, caso haja uma falha permanente no servidor, os dados também serão permanentemente perdidos.

Com objetivo de contornar o desafio descrito, o BookKeeper foi concebido, um serviço de *logging* que, para assegurar a durabilidade de uma aplicação, dissocia a disponibilidade do *log* da disponibilidade do servidor, enquanto implementa a replicação do *log*.

2.3.1 Arquitetura

Arquitetura do Apache BookKeeper possui três abstrações, *Ledger*, *Bookie* e *Client*, as quais são descritas abaixo (JUNQUEIRA; KELLY; REED, 2013):

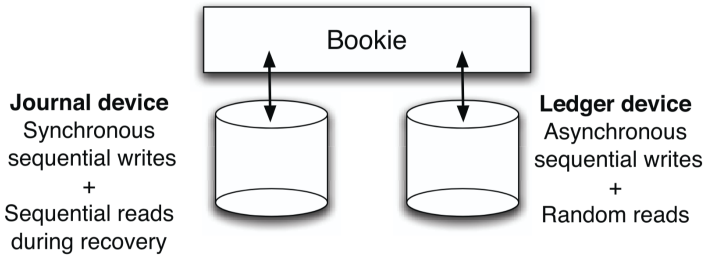
- O BookKeeper usa o termo "*ledgers*" para denotar os *logs* e "*entries*" para denotar os registros de um *log*. *Ledger* é unidade primitiva do BookKeeper e essencialmente uma sequência finita de *entries*, cada *entry* é uma sequência de *bytes* juntamente com outros metadados, como o identificador do registro e o identificador do *ledger* ao qual o registro pertence.

- O BookKeeper replica e persiste os dados através de múltiplos servidores, estes são denominados "*bookies*". A Figura 5 ilustra a arquitetura de um *bookie*, o qual utiliza dois dispositivos, idealmente em dois discos físicos distintos, um *journal device* e um *ledger device*. O *journal device* atua como um *write-ahead log* (WAL) e o *ledger device* contém uma cópia indexada de um fragmento de *ledger*, que o *bookie* usa para as solicitações de leitura.

[2] https://bookkeeper.apache.org

- Uma aplicação deve utilizar o BookKeeper *client* para interagir com o serviço. O *client* é responsável por processar as solicitações da aplicação, submetê-las aos *bookies* e retornar o resultado das operações para a aplicação.

Figura 5 – Arquitetura de um *bookie*



Fonte: Junqueira, Kelly e Reed (2013)

O BookKeeper foi projetado para possibilitar o processamento concorrente de *logs* do tipo *single-writer/multiple-reader*, ou seja, assume que apenas um cliente é responsável por registrar *entries* no *ledger*. Eventualmente, o *ledger* é fechado e, a partir deste instante, opera apenas em modo *read-only*. Essa restrição possibilita ao BookKeeper aplicar um protocolo de quórum simples, em que para tolerar até f falhas, são necessárias $f+1$ *bookies* na confirmação de um registro.

O BookKeeper usa três parâmetros para definição de quórum, um *ledger ensemble* (E), um *write* quórum (q_w) e um *ack* quórum (q_a). O ensemble é o número de *bookies* em que o ledger será persistido, o *write* quórum é o número de réplicas que uma requisição deve ser registrada, já o *ack* quórum é o número de réplicas que deverão confirmar o registro. Ao instanciar um *ledger* deve-se manter a invariante $E >= q_w >= q_a$. Com essa configuração, o BookKeeper pode tolerar até $q_a - 1$ falhas para que o serviço permaneça operante (The Apache Software Foundation, 2025a).

2.3.2 Protocolo do BookKeeper

Para assegurar a consistência forte, ao adicionar registros ao *ledger*, o BookKeeper aplica de uma variação do *two-phase commit* (2PC), o *LastAddConfirmed* (LAC). Este mecanismo permite que um *writer* envia um lote de registros, incrementando um contador designado como *LastAddPushed* (LAP), e receba confirmações quando estas são efetivamente confirmadas, incrementando o LAC (GUO; DHAMANKAR; STEWART, 2017; JUNQUEIRA; KELLY; REED, 2013).

O *writer* do *ledger*, contudo, pode falhar prematuramente antes de fechar o *ledger*. Nessa situação, como não é possível determinar qual é o último registro válido do *ledger*, um procedimento de recuperação denominado *ledger recovery* é aplicado. O objetivo central da recuperação é determinar qual foi o último registro que foi corretamente replicado para um quórum de *bookies* e, em seguida, registrar essa informação no ZooKeeper. Para determinar o último registro, o BookKeeper *client* solicita todos os registros a partir do *LastAddConfirmed* (LAC) para os *bookies* e, a partir dessa sequência, é possível determinar qual o último registro que foi corretamente replicado entre o quórum de *bookies*.

Ao iniciar o processo de *ledger recovery* há a possibilidade que o *writer* prossiga com a adição de outros registros, todavia, para prevenir esse cenário, é empregado um mecanismo de *fencing*. Essa prevenção é vital dentro do BookKeeper, pois evita que o *writer* registre uma *entry* com ID superior ao ID que o procedimento de recuperação está prestes a determinar como o fim do *ledger*. O *fencing*, portanto, impede que operações de adição sejam executadas assim que o procedimento de recuperação do *ledger* é iniciado. A solução envolve a marcação dos *bookies* com uma *flag* que os impede de executar requisições de adição para o *ledger* que está sob processo de recuperação.

Para a gerência dos *ledgers*, o BookKeeper consta com vários metadados, como os detalhes sobre a composição do *ledger ensemble*, *write* quórum, *ack* quórum, status do ledger (*OPEN*, *CLOSED* ou *IN_RECOVERY*), etc. Como discutido, o BookKeeper também registra a última *entry* de um *ledger* fechado como parte de seus metadados. O BookKeeper delega a responsabilidade de gerência destes metadados para o ZooKeeper, embora a implementação do BookKeeper seja capaz de usar qualquer armazenamento que suporte uma primitiva *compare-and-swap* (CAS) (JUNQUEIRA; KELLY; REED, 2013; The Apache Software Foundation, 2025a).

2.3.3 Ledger API

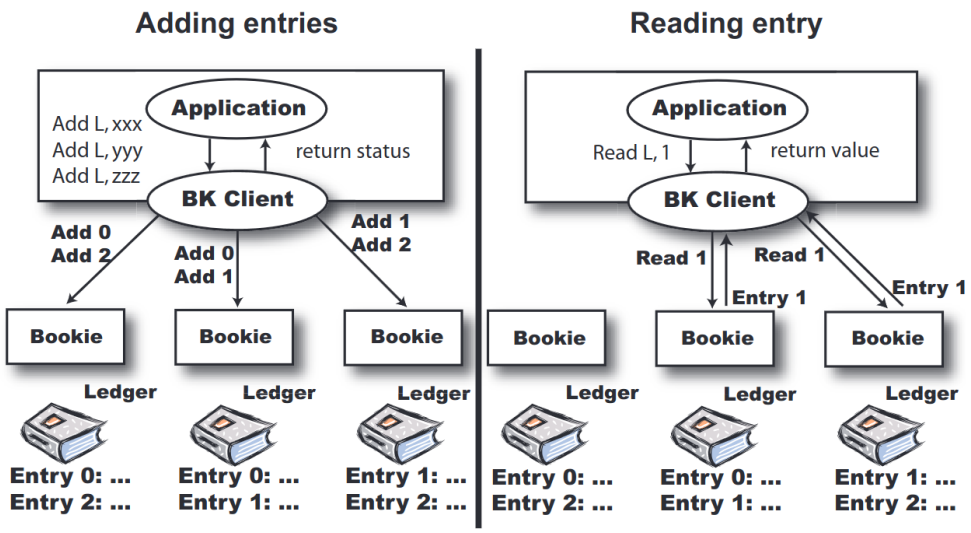
A Ledger API é uma API de baixo nível do BookKeeper que permite a interação direta com os *ledgers* do BookKeeper. Ela proporciona, em suma, dois papéis para manipulação dos *ledgers*: *writer* e *reader*. Uma aplicação inicialmente irá designar um *writer* que será responsável por criar um *ledger* e por anexar registros a ele. Eventualmente, esse *writer* é encerrado e o conteúdo do *ledger* torna-se imutável e apenas *readers* são permitidos (JUNQUEIRA; KELLY; REED, 2013). A API provê algumas operações para a manipulação dos *ledgers*, dentre elas destaca-se as operações listadas:

- Criar um *ledger*;

- Anexar *entries* ao ledger;
- Solicitar um *ledger* para leitura, esteja ele encerrado ou com um *writer* associado;
- Solicitar uma série de *entries* de um *ledger* ou uma *entry* em específico;
- Fechar um *ledger* para impedir a adição de novas *entries*;
- Excluir um *ledger* em sua totalidade.

Para uma aplicação criar um *ledger* ela deve indicar o *ensemble ledger* (E), *write* quórum (q_w) e *ack* quórum (q_a). Caso haja sucesso na criação de um *ledger*, um *writer* também é registrado, o qual é aplicado para adicionar *entries* exclusivamente a este *ledger*. Para cada registro que o *writer* adiciona ao *ledger*, ele replica o registro por q_w réplicas e aguarda a confirmação de q_a réplicas. O *reader* segue o mesmo fluxo, para ler um determinado registro e de um *ledger*, o *reader* determina as q_a réplicas em que o registro e foi replicado e o solicita.

Figura 6 – Operações do BookKeeper



Fonte: Junqueira, Kelly e Reed (2013)

Essas operações no BookKeeper estão ilustradas na Figura 6, dado $E = 3$, $q_w = 2$ e $q_w = 2$. Para adição, uma aplicação, através do BookKeeper *client* (BK Client), adiciona três registros a um determinado *ledger* e cada registro é replicado por $f + 1$ *bookies*. Nesse processo, o BookKeeper alterna os quóruns através do método *round-robin* para distribuir a carga uniformemente entre as réplicas. Para a leitura, a aplicação solicita ao BK Client um determinado registro do *ledger* e este determina os $f + 1$ *bookies* em que o registro foi replicado para retornar o valor.

3 REVISÃO BIBLIOGRÁFICA

Visando a construção de uma perspectiva sobre o estado da arte da replicação transparente, foi conduzida uma revisão bibliográfica, a qual selecionou algumas obras para compor os trabalhos correlatos desta monografia. Os trabalhos correlatos são categorizados a partir de abordagens distintas dentro da replicação. Inicialmente é discutido a utilização de *logs* distribuídos para a construção de máquinas de estado replicadas. Outra seção discute softwares com replicação integrada que são disponibilizados como um SaaS (*Software as a Service*). A última seção explora, mais especificamente, replicação baseada em *middleware*, tal qual o propósito desse trabalho.

3.1 *LOGS* DISTRIBUÍDOS

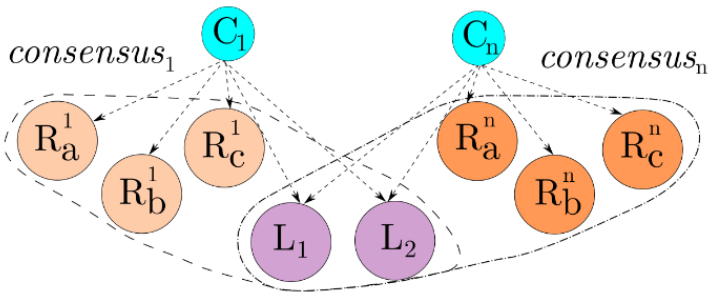
Replicação de Máquinas de Estados (RME) é uma abordagem altamente aplicada no desenvolvimento de sistemas tolerantes a falhas, porém a complexidade envolvida na implementação dessa estratégia exige dos projetistas conhecimentos não-triviais de tolerância a falhas. Nessa perspectiva, Xavier et al. (2020) propõe desacoplar os *logs* das réplicas e proporcioná-los com um serviço anexado a arquitetura, com objetivo de simplificar o desenvolvimento de técnicas de *logging*, incentivar o compartilhamento de recursos e reduzir os custos associados a replicação em infraestruturas *pay-per-use*.

O projeto é construído, portanto, através da disponibilização de um serviço de *logs* independente que é incorporado na arquitetura de uma aplicação replicada, na qual o serviço compartilha o protocolo de consenso com as demais réplicas. A Figura 7 ilustra essa estratégia, em que o processo de *logging* é deslocado das réplicas (R^n) para serviços externos (L_1 e L_2). O serviço de *logs* é, por exemplo, incorporado como um *learner*, para implementações do Paxos, ou *non-voters*, para implementações do Raft. Essa desassociação, permite que as réplicas atuem apenas no processamento das solicitações dos clientes, enquanto o serviço atua no registro da sequência de solicitações.

Para a avaliação do serviço de *logs* desacoplados foram desenvolvidas duas aplicações, um *kvstore* (*key-value store*) e um *diskstore* (serviço de diretório). Para cada aplicação, configurou-se três versões: uma com o serviço de *logs* desabilitado, uma na qual a aplicação é responsável pelo registro dos *logs* e outra com o serviço de *logs* desacoplado.

Inicialmente, a avaliação focou na latência e vazão de cada aplicação à medida que o número de clientes era gradualmente incrementado. Para o *kvstore* e *diskstore*, a aplicação sem *logs* obteve um desempenho superior, um resultado já previsto, visto que a adição de *logs* naturalmente eleva a latência da aplicação. Já as versões com *logs* não divergiram muito em

Figura 7 – Proposta de *logging* para Replicação de Máquina de Estados



Fonte: Xavier et al. (2020)

questões de desempenho e, apesar de não apresentar benefícios claros com apenas uma aplicação, os resultados com várias aplicações compartilhando um único serviço de *logs* apresentou uma melhor distribuição na utilização dos recursos e, consequentemente, gerou redução de custos em infraestruturas *pay-per-use*.

Apesar de a estratégia de desacoplamento auxiliar no desenvolvimento, ela desacopla apenas o processo de *logging* e, consequentemente, ainda exige a atuação das réplicas no fluxo de replicação. Neste contexto, o projeto prevê o desacoplamento integral, possibilitando que as réplicas atuem apenas no processamento das operações solicitadas por seus clientes.

Balakrishnan et al. (2013) também propõe o CORFU, um *log* distribuído e compartilhado que disponibiliza abstrações para certificar consistência forte em sistemas distribuídos, mesmo diante de falhas e assincronia. Assim, o CORFU possibilita a construção de sistemas distribuídos com exigência de desempenho e consistência forte, como banco de dados, *key-value store*, máquinas de estado replicadas e serviços de metadados.

Tradicionalmente, para escalar sistemas de armazenamento de dados, são aplicadas estratégias de fragmentação (*sharding*) no banco de dados, como é o caso do Percolator (PENG; DABEK, 2010), Megastore (BAKER et al., 2011) e Spanner (CORBETT et al., 2013). Porém, há a exigência de mecanismos dispendiosos, como bloqueio de duas fases ou gerenciadores de concorrência centralizados, para prover consistência forte entre as partições. O CORFU visa superar o *tradeoff* entre desempenho e consistência. Ele é projetado para escalar para centenas de milhares de operações concorrentes por segundo, preservando a semântica de cópia única.

Internamente, o CORFU é distribuído por um *cluster* de máquinas, em que os dados são totalmente replicados para garantir a tolerância a falhas, sem recorrer a fragmentação de dados (*sharding*) ou ao sacrifício da consistência. O CORFU permite que os clientes adicionem (*append-to*) dados ao final de um *log* compartilhado e leiam (*read-from*) dele por meio de uma rede. Uma característica central do CORFU é que não há um único gargalo de E/S para estas operações.

A relevância do CORFU para esta monografia reside no fato que ele implementa um serviço de ordenação total tolerante a falhas, atuando como um mecanismo de consenso durável e escalável. Nessa perspectiva, o CORFU é um forte candidato para o serviço de *log* distribuído do Middle-earth, visto que provê durabilidade e consistência forte (mais detalhes quanto a este módulo do Middle-earth são apresentados na Subseção 4.1.1).

3.2 *SOFTWARE AS A SERVICE*

Um Sistema de Gerenciamento de Banco de Dados (SGBD) pode ser classificado como tradicional ou nativo da nuvem. Soluções tradicionais, como o Oracle[1], MySQL[2] e PostgreSQL[3], têm o desempenho afetado pela aplicação de estratégias como *write-ahead logging* (WAL) e *checkpoint*. Em vista disso, há o esforço para a construção de bancos de dados nativos da nuvem, como o Google Spanner (CORBETT et al., 2013) e o Amazon Aurora (VERBITSKI et al., 2017), serviços altamente disponíveis e escaláveis como parte do portfólio das plataformas de nuvem (NEU et al., 2019).

O Spanner é um sistema de gerenciamento de dados distribuído. Concebido para ser escalável, multiversão, distribuído globalmente e replicado de modo síncrono. Inicialmente, o Spanner foi projetado como um banco de dados chave-valor que oferecia transações *multi-row*, consistência forte e *failover* transparente entre *datacenters*. Ao longo dos anos, o Spanner evoluiu para um sistema de banco de dados relacional (CORBETT et al., 2013; BACON et al., 2017).

A motivação para essa evolução foi a dificuldade dos projetistas do Google de construir aplicações OLTP (*Online Transaction Processing*) sobre um banco de dados chave-valor que carecia de transações *multi-row*, replicação consistente e, principalmente, de uma linguagem de consulta robusta. Neste contexto, como resultado, o Spanner foi transformado em um serviço com suporte a sintaxe SQL, com a execução de consultas rigidamente integrada com as outras características arquiteturais do Spanner, como consistência forte e replicação global.

As transações do Spanner utilizam um *write-ahead redo log* replicado e o protocolo de consenso Paxos para que haja um acordo quanto ao conteúdo do *log*. Cada fragmento do banco de dados é atribuído a exatamente um grupo Paxos. Um determinado grupo pode ter vários fragmentos atribuídos. Assim, a confirmação de um registro no *log* requer a replicação para um quórum de réplicas.

Entretanto, a compatibilidade com bancos de dados tradicionais é crucial, já que há

[1] https://www.oracle.com/database
[2] https://www.mysql.com/
[3] https://www.postgresql.org/

várias aplicações legadas ainda dependentes destas soluções. Isso vai além do suporte à mesma sintaxe SQL disponibilizado pelo Spanner, visto que há inúmeras extensões aplicáveis aos sistemas tradicionais que são configuradas apenas nos servidores originais. O Amazon Aurora visa preencher essa lacuna. Um serviço construído sobre o MySQL que assegura compatibilidade com uma solução tradicional, enquanto desloca o armazenamento para a nuvem e o replica por vários nós para possibilitar alta escalabilidade e resiliência (VERBITSKI et al., 2017).

Amazon Aurora possui como núcleo de desenvolvimento o MySQL Community Database Engine e adota a premissa de que o "*log* é o banco de dados" e, portanto, aplica um serviço de *logs* distribuídos para assegurar durabilidade e consistência entre as réplicas, no lugar de estratégias tradicionais, como *write-ahead logging* (WAL) e *checkpoint*. O Amazon Aurora aplica uma solução *primary-backup*, na qual a réplica mestre é responsável por registrar os *logs* e outras as réplicas irão reconstruir os dados a partir desses *logs* disponíveis.

Para a avaliação do desempenho foi comparado a vazão do Amazon Aurora com o MySQL 5.6 e MySQL 5.7 em cinco instâncias EC2 (*Elastic Compute Cloud*) da família r3 (large, xlarge, 2large, 4xlarge, 8xlarge). O Amazon Aurora obteve o dobro de vazão em todas as instâncias, com exceção da instância 8xlarge, na qual obteve uma vazão 5 vezes superior ao MySQL 5.7.

Como supracitado, o Google Spanner e Amazon Aurora são construídos para proporcionar alta disponibilidade e escalabilidade, a partir de estratégias de replicação transparente. O Spanner, apesar de ofertar suporte a sintaxe do SQL, carece de compatibilidade com bancos de dados tradicionais. Já o Amazon Aurora preenche essa lacuna ao construir um serviço sobre o MySQL. Entretanto, como o Amazon Aurora é construído apoiado em um *fork* do MySQL, ele possui uma forte dependência do código do MySQL e, portanto, é complexo quanto à manutenção e incorporação de novas funcionalidades. O Amazon Aurora e o Google Spanner são, ainda, altamente acoplados a outros serviços de suas plataformas e ofertados apenas como um *Software as a Service* (SaaS).

3.3 REPLICAÇÃO BASEADA EM *MIDDLEWARE*

Os bancos de dados nativos da nuvem, como o Google Spanner (CORBETT et al., 2013) e o Amazon Aurora (VERBITSKI et al., 2017), diferentemente das soluções tradicionais, proveem alta disponibilidade ao proporcionar replicação transparente e escalabilidade automática. Entretanto, a compatibilidade com sistemas legados é crucial, indo além do suporte à mesma sintaxe. Embora o Amazon Aurora proporcione essa compatibilidade com o MySQL, ele gera uma sobrecarga operacional associada à manutenção e à incorporação de novas funcionalidades conforme estas são introduzidas ao MySQL.

Neste contexto, Coelho et al. (2023) propõem o Loom, um sistema desagregado que proporciona generalização ao aplicar o servidor original na construção do ecossistema de replicação. O Loom implementa dois componentes entre a aplicação e o banco de dados, o *Loom Client* e *Loom Server*, sem comprometer a estrutura original dos servidores, além de delegar a responsabilidade da durabilidade para um serviço de *logs* distribuídos externo.

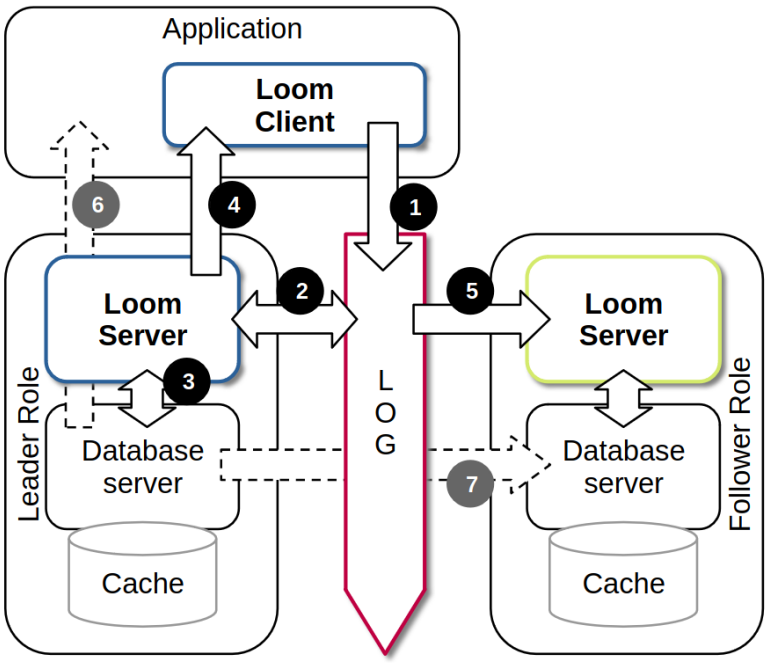
O *Loom Client* é responsável por fornecer uma interface de comunicação para a aplicação com o serviço de replicação e por registrar as transações no serviço de *logs*. Essa interface é desenvolvida acima do JDBC (*Java Database Connectivity*) e, portanto, possibilita a portabilidade de aplicações sem modificações. O serviço de *logs* selecionado é o Apache BookKeeper, através da interface DistributedLog, que provê *logs* distribuídos com replicação, durabilidade e consistência forte através de uma API *Write-Read Proxy*. Já o *Loom Server* é responsável diretamente pela comunicação com o servidor ao submeter as transações disponíveis no serviço de *logs*. Após o processamento da transação, os resultados da execução são enviados para a aplicação através do *Loom Server* e *Client*.

O Loom está ilutrado na Figura 8, na qual é possível vislumbrar o banco de dados (*Database Server*), o *Loom Server*, o *Loom Client* e o serviço de *logs* (LOG). O *Loom Client* é integrado a uma aplicação e é responsável por anexar as transações no serviço de *logs* (1). A transação é resgatada pelo *Loom Server* (2) e submetida ao *Database Server* (3). O resultado da execução é enviado para a aplicação através do *Loom Server e Client* (4), enquanto as atualizações são persistidas no *log* distribuído (5). Aplicações *read-only* têm a possibilidade de conexão direta com o *Database Server* (6). Para facilitar a recuperação e reconfiguração, *caches* são atualizadas diretamente entre os servidores, recorrendo ao serviço de *logs* apenas quando necessário (7).

Para a avaliação do Loom houve a configuração de três cenários distintos: o primeiro compreende um único servidor para uma avaliação preliminar da arquitetura; o segundo foi implantado na *Google Cloud Platform*, com objetivo de avaliar a portabilidade e escalabilidade do Loom; o terceiro cenário foi configurado em um HPC privado, com objetivo de mensurar a vazão máxima. O *benchmarking* demonstrou que o Loom proporciona uma vazão 150x maior em comparação com a replicação nativa do PostgreSQL. Já no cenário configurado para HPC, o Loom ainda apresentou uma vazão superior ao cenário com um único servidor, evidenciando a capacidade de escalabilidade do Loom.

O Loom, diferentemente do Amazon Aurora, propõe um sistema completamente desagregado do servidor original e possibilita que a replicação seja estruturada independentemente do SGBD. Porém, apesar de possibilitar generalização quanto ao SGBD aplicado à infraestrutura, o Loom, tal qual o Amazon Aurora, está focado em um único contexto e não proporciona

Figura 8 – Arquitetura do Loom



Fonte: Coelho et al. (2023)

um sistema totalmente independente da aplicação. Em vista disso, este trabalho visa contornar essa limitação ao propor um serviço independente, que possa ser empregado na configuração da replicação de qualquer aplicação.

3.4 DISCUSSÃO

A comparação das soluções apresentadas nos trabalhos correlatos com as características que a solução proposta visa propiciar está ilustrada na Tabela 1. As propostas de Xavier et al. (2020) e Balakrishnan et al. (2013) visam desacoplar apenas o processo de *logging* e, portanto, são parcialmente modulares, já que irão desacoplar apenas uma porção do processo de replicação. Essas soluções, tal qual a proposta desse trabalho, são independentes da aplicação e, portanto, são soluções compartilháveis entre várias aplicações. A solução de Xavier et al. (2020) foi o desenvolvimento de uma prova de conceito e não apresentou demonstrações concretas quanto a sua escalabilidade. As propostas de Bacon et al. (2017) e Verbitski et al. (2017) são soluções que visam disponibilizar um software completo ao usuário, no qual a plataforma gerencia todo o processo de replicação. Assim, essas soluções estão presentes no tópico de replicação transparente, porém, não são soluções modulares e tão pouco compartilháveis com outras aplicações, já que são disponibilizadas como um SaaS (*Software as a Service*) nos portfólios das plataformas. A proposta de Coelho et al. (2023) segue a mesma abordagem

dessa monografia ao proporcionar replicação baseada em *middleware*. Entretanto, essa solução foi desenvolvida para o contexto de banco de dados e, portanto, não proporciona uma solução totalmente independente do contexto da aplicação.

Tabela 1 – Comparação entre as soluções dos trabalhos correlatos

Autor(es)	**Proposta**	**Modularidade**	**Independência**	**Escalabilidade**
Xavier et al. (2020)	Serviço de log desacoplado para a implementação de replicação	Sim/Não	Sim	Não
Balakrishnan et al. (2013)	CORFU: Log distribuído e compartilhado	Sim/Não	Sim	Sim
Bacon et al. (2017)	Banco de dados distribuído	Não	Não	Sim
Verbitski et al. (2017)	Banco de dados distribuído	Não	Não	Sim
Coelho et al. (2023)	Middleware para replicação de banco dados	Sim	Não	Sim

Fonte: Elaborado pela autora

4 MIDDLE-EARTH

Este capítulo propõe o Middle-earth, um serviço de replicação como *middleware* responsável por atuar como interceptador das requisições para uma determinada aplicação e, apenas após assegurar a ordenação e durabilidade das requisições, transferi-las para as réplicas disponíveis. Além disto, ele é responsável por coordenar a devolutiva da requisição para o cliente, após o processamento por quaisquer réplicas. Ao desacoplar a responsabilidade de replicação das réplicas o Middle-earth visa proporcionar alta modularidade, independência, escalabilidade e otimizações na implementação de replicação aos projetistas. As seções subsequentes detalham, respectivamente, a concepção do Middle-earth e a implementação de uma prova de conceito (POC, do inglês *proof of concept*) dele.

4.1 ARQUITETURA

A arquitetura proposta visa proporcionar um projeto modular ao assegurar que toda a estratégia de replicação das réplicas seja deslocada para um serviço independente, possibilitando que as réplicas atuem apenas no processamento das solicitações de seus clientes. Ao construir este serviço independente, o Middle-earth também possibilita o compartilhamento da infraestrutura por aplicações distintas e a possibilidade de alocar recursos específicos apenas para o *middleware*. Em vista disso, é possível operar as réplicas a partir de estratégias *serverless*, sem o comprometimento da aplicação, já que a durabilidade das requisições é assegurada pelo Middle-earth.

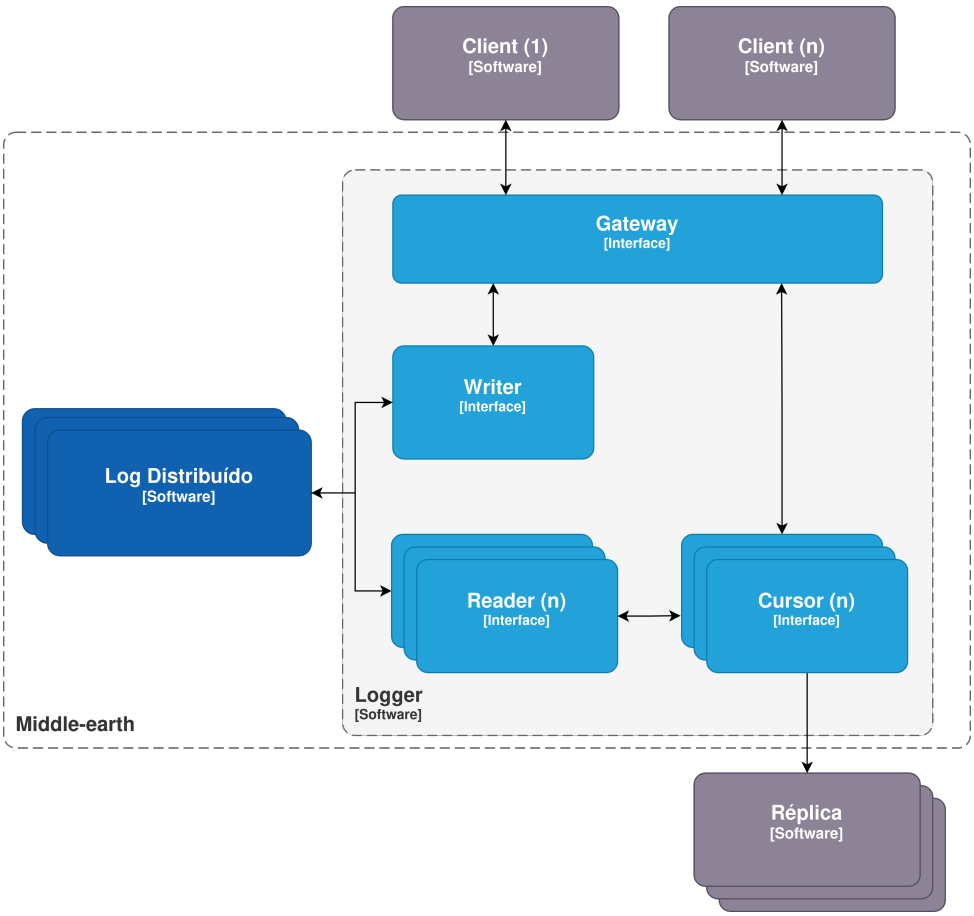
A arquitetura do projeto, representada na Figura 9, ilustra clientes e réplicas de uma determinada aplicação e o Middle-earth, este composto por dois serviços, um *log* distribuído e o *logger*. O *logger*, por sua vez, é constituído por outros módulos, os quais serão descritos em uma seção específica para o *logger*. A arquitetura é projeta pela perspectiva da Replicação de Máquinas de Estado (RME), em que as réplicas irão avançar por estados idênticos ao processar de modo ordenado operações determinísticas. Por consequente, compreende-se que apenas uma *thread* deve ser operativa nas réplicas da aplicação, visto que o paralelismo é, geralmente, não-determinístico. Como mencionado no Subseção 2.1.1, há propostas de Replicação de Máquinas de Estado Paralela, entretanto, para a elaboração do Middle-earth, tal estratégia não foi avaliada, devido a necessidade de compreensão da semântica da aplicação a ser replicada, distanciando-se da proposta do Middle-earth de proporcionar alta modularidade.

As réplicas da aplicação são responsáveis apenas pelo processamento das requisições dos clientes a partir da ordenação estabelecida pelo *middleware*. Consequentemente, desde que haja uma réplica ativa, é possível processar as requisições dos clientes. Isso permite que o

Middle-earth seja resiliente a possíveis falhas nas réplicas e permitindo operar, consequentemente, com $f + 1$ réplicas, para tolerar até f falhas.

A seguir são descritos o *log* distribuído, o *logger* e a colaboração entre eles para a concepção do Middle-earth.

Figura 9 – Middle-earth com clientes e réplicas de uma aplicação determinada alvo



Fonte: Elaborado pela autora

4.1.1 Log distribuído

O *log* distribuído é responsável por assegurar a durabilidade, disponibilidade e consistência forte dos dados. O *log* estabelece comunicação com o *logger* a partir de dois módulos internos ao *logger*, o *Writer* e o *Reader*, estes detalhados na subseção do *logger*. O *log* é independente dentro do *middleware* e, portanto, é possível implementá-lo a partir de estratégias de replicação, como a replicação ativa ou passiva (SCHNEIDER, 1990; SCHNEIDER, 1993; BUDHIRAJA et al., 1993), ou aplicar um serviço que ofereça *logs* distribuídos, como o Apache BookKeeper e CORFU (The Apache Software Foundation, 2025a; BALAKRISHNAN et al.,

2013). Desde de que o *log* selecionado supra as exigências do módulo é possível utilizá-lo na implantação do Middle-earth.

Para assegurar a durabilidade dos dados do Middle-earth, deve-se configurar réplicas para o *log* distribuído, cujo valor depende da estratégia adotada. Contudo, a política comumente aplicada em sistemas distribuídos é $2f + 1$ réplicas, para toleras até f falhas.

Apesar do *log* distribuído exigir a instância de um conjunto de servidores, essa questão é compensada pela viabilidade de operar as réplicas com um quórum simples. Além da otimização de recursos ao deslocar a lógica de replicação das réplicas, possibilitando que elas atuem apenas na lógica da aplicação.

4.1.2 Logger

O *logger* é responsável pela intercepção das requisições, delegar a durabilidade para o *log* distribuído e, por fim, pela coordenação das requisições entre os clientes e as réplicas de uma aplicação. O *logger* conta com alguns módulos internos para a execução destas atribuições: *Gateway*, *Writer*, *Reader* e *Cursor*.

O módulo *Gateway* irá representar um *wrapper* da *Application Programming Interface* (API) de alguma determinada aplicação, seu objetivo é abstrair para o cliente que há um *middleware* entre a comunicação dele e do servidor. O *Gateway* é responsável, portanto, por efetivamente interceptar as requisições dos clientes, encaminhá-las para o *Writer* e, após a confirmação de que a requisição está devidamente registrada no *log*, notificar os *Cursors* que existem requisições pendentes para as réplicas. Após o processamento da requisição por qualquer uma das réplicas ele executa a devolutiva para o cliente.

O *Writer* abstrai interface entre o *logger* e o *log* distribuído, ele é responsável por disponibilizar os métodos necessários para o registro no *log*. O *Reader*, tal qual o *Writer*, abstrai a comunicação com o *log* distribuído e, portanto, deve disponibilizar os métodos necessários para a leitura do *log*.

O *Cursor* é responsável por encaminhar as requisições pendentes para as réplicas disponíveis após a notificação do *Gateway*. Um *Cursor* é responsável por apenas uma réplica e estabelece comunicação com o *Reader* para a solicitação dos registros no *log*. Após o processamento da requisição pela réplica, o *Cursor* deve encaminhar para o *Gateway* a devolutiva.

O *logger* foi desenvolvido para atender a uma aplicação específica, visto que o *Gateway* e o Cursor são abstrações, tanto para o cliente quanto para as réplicas, de que há um *middleware* intermediando a comunicação entre eles. Assim, caso seja necessário atender a distintas aplicações, será necessário desenvolver *Gateway* e *Cursor(s)* dado o padrão de comunicação adotado pela aplicação, como *socket*, HTTP, gRPC ou até APIs mais específicas, como

o JDBC. Neste cenário, será necessário desenvolver uma biblioteca que deverá ser incorporada ao cliente como uma abstração da biblioteca padrão.

Caso haja necessidade de atender a aplicações distintas em uma única infraestrutura do Middle-earth, será necessário configurar um *logger* por aplicação. Enquanto o *log* distribuído pode ser compartilhado entre as diferentes aplicações. Essa configuração disponibiliza flexibilidade ao *middleware* para alocar recursos específicos entre os módulos.

Como o *logger* e o *log* distribuído são independentes dentro do *middleware*, a arquitetura possibilita que o Middle-earth seja escalável por quantia de réplicas e por quantia de aplicações distintas vinculadas a infraestrutura. Essa ação é possível visto que a quantidade de réplicas não têm uma interferência direta no processo de durabilidade dos dados e coordenação das requisições, não interferindo, portanto, na escalabilidade do *logger*. Além disto, como o *log* distribuído pode ser escalado independentemente, é possível adicionar aplicações diferentes ao Middle-earth a partir da instância de outros *loggers*.

4.2 IMPLEMENTAÇÃO DO MIDDLE-EARTH

Para a implementação de uma POC do Middle-earth foi aplicado o Apache BookKeeper, em sua versão 4.17.1 (The Apache Software Foundation, 2024a) como *log* distribuído. Também foi utilizado o Apache ZooKeeper, em sua versão 3.8.4 (The Apache Software Foundation, 2024b), visto que este é responsável pela gerência dos metadados, tanto do BookKeeper, quanto do próprio Middle-earth. Para o desenvolvimento do *middleware*, foi selecionado a linguagem de desenvolvimento Java, mais especificadamente o Java 17 (Oracle Corporation, 2025). O Java foi eleito devido a dependência da aplicação ao Apache BookKeeper e Apache ZooKeeper, cujas as APIs estão disponíveis em Java, e a disponibilidade de artefatos para a coordenação de tarefas concorrentes. Ademais, foi adotada a comunicação via *Sockets* para o *middleware*, devido a facilidade de implementação deste mecanismo.

Esta seção detalha a implementação dessa configuração do Middle-earth, na qual, são apresentados o *log* distribuído, as interfaces do *logger* e a implementação destas para a concepção do *middleware*.

4.2.1 Log distribuído

Para o *log* distribuído, selecionou-se o Apache BookKeeper, um serviço que provê replicação, durabilidade e consistência forte e, portanto, supre os requisitos necessários do componente. A escolha do BookKeeper, em detrimento de uma implementação própria ou de outros serviços de *logs* distribuídos, justifica-se pela simplicidade de sua integração com a aplicação

e por ser um *log* consolidado dentro da indústria, apresentando uma documentação acessível e, principalmente, uma comunidade ativa.

Para a implantação do Apache BookKeeper é requerido, inicialmente, a configuração do Apache ZooKeeper, já que este é responsável pela coordenação dos metadados do BookKeeper. Como o ZooKeeper pode operar no modo *standalone*, é possível implantá-lo com apenas um servidor, contudo, também é possível configurar réplicas do serviço.

Além do ZooKeeper operar na gerência dos metadados do BookKeeper, ele é aplicado pelo Middle-earth para coordenar outras informações associadas ao *log*, como uma identificação do *log* dentro do *middleware* para que ele possa operar com uma abstração single-log. Como um *ledger* do BookKeeper não possibilita a adição de *entries* após seu fechamento, há necessidade de coordenar quais *ledgers* estão associados a um *log* através do ZooKeeper para possibilitar essa abstração. Outro metadado coordenado pelo ZooKeeper é o último registro lido por cada réplica, essa ação é necessária para, caso o *logger* seja acometido por uma falha, seja possível prosseguir a partir desse registro quando o *logger* for substituído.

Na inicialização de um *writer* do BookKeeper é necessário informar, além da quantidade de *bookies* disponíveis (*ledger ensemble*), o *write* quórum e um *ack* quórum. O *write* quórum é o número de réplicas que uma requisição deve ser registrada, já o *ack* quórum é o número de réplicas que deverão confirmar o registro. Após uma solicitação de um registro para o BookKeeper, o BookKeeper *client* repassa a solicitação para o *write* quórum definido e aguarda pelo *ack* quórum para confirmar o registro.

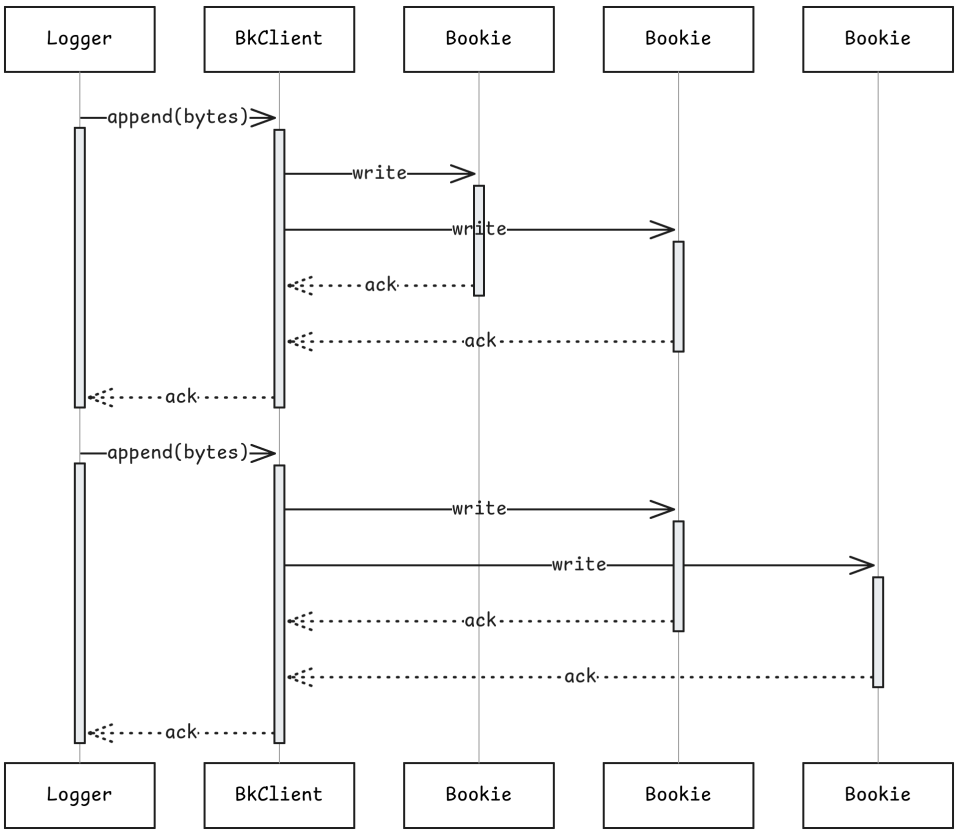
O diagrama, retratado na Figura 10, amplia a comunicação do *logger* com os *bookies* através da interface BookKeeper *client* (BkClient). Para essa ilustração, assume-se um *ledger ensemble* (E), um *write* quórum (q_w) e um *ack* quórum (q_a) de, respectivamente, $E = 3$, $q_w = 2$ e $q_a = 2$. Nessa configuração, o BookKeeper *client* irá registrar sempre em dois *bookies* e esperar pela confirmação destes mesmos dois *bookies* para efetivar o registro de um dado.

4.2.2 Logger

Devido a configuração do Apache BookKeeper que possibilita apenas um *writer* por *log*, o *logger* ficou limitado a uma única instância (réplica) por aplicação. Essa configuração torna o *logger* um ponto único de falha (SPOF, do inglês *Single Point of Failure*) na infraestrutura, todavia, na seção Seção 6.1 são discutidas estratégias para contornar essa limitação do Middle-earth.

Para refletir os módulos discutidos na arquitetura, o *logger* foi desenvolvido através de interfaces extensíveis, assegurando, assim, a possibilidade de aplicar outro *log* distribuído e permitir a utilização do *logger* por quaisquer aplicações, desde que haja a implementação de

Figura 10 – Comunicação entre o *looger* e os *bookies* do BookKeeper



Fonte: Elaborado pela autora

um *wrapper* dado a API estabelecida pela aplicação. A seguir, são apresentadas as interfaces e a implementação destas para a construção do Middle-earth.

4.2.2.1 Interfaces

A interface *Gateway* define apenas os métodos $init()$ e $close()$, responsáveis, respectivamente, pela inicialização e finalização do *wrapper*. Não são especificados outros métodos, visto que a interface deve proporcionar flexibilidade para diferentes aplicações.

Entry é uma abstração de um registro no *log* e possui dois atributos, $entryId$ e $payload$ responsáveis, respectivamente, por informar o identificador *ID* e o *payload* do registro no *log* distribuído.

A interface *Writer* possui duas variações do método *write*, o $write(byte[]\ data)$ e o $write(byte[]\ data,\ Callback\ callback)$. Enquanto um é responsável por apenas registrar uma determinada sequência de *bytes* no *log* distribuído, o outro é responsável por invocar uma função *callback* informada após a confirmação da inserção do registro. O *Writer* também disponibiliza um método $getReader()$, responsável por instanciar um *Reader* do *log*.

O *Reader* possui o método $read(long\ entryId)$, que irá informar a *Entry* associada ao *ID* solicitado, e o $read(long\ firstEntryId, long\ lastEntryId)$, responsável por informar as *entries* dentro da faixa de *IDs* solicitados.

A interface *Cursor* possui apenas os métodos $notifyEntryAvailable(entryId)$ e $close()$, visto que, tal qual o *Gateway*, é dependente da aplicação e deve permitir flexibilidade para outras implementações. O método $notifyEntryAvailable(entryId)$ é invocado pelo *Gateway* sempre que um registro é inserido no *log* distribuído e é responsável por transferir os registros para as réplicas.

4.2.2.2 Implementação

Para a implementação do Middle-earth foi aplicado o Apache BookKeeper como *log* distribuído e adotada uma comunicação via *Sockets* entre os clientes e as réplicas. Assim, a implementação das interfaces discutidas prossegue dadas tais premissas.

O *SocketGateway* (implementação do *Gateway*) foi implementado conforme os padrões de comunicação via *Sockets*. A implementação do método $init()$, ilustrada no Algoritmo 1, é responsável por instanciar um *Socket* (Linha 7), para comunicação do *middleware* com os clientes da aplicação. Além de inicializar o *Socket*, o método é responsável por instanciar um *Writer* para o *log* (Linha 8) e pela inicialização dos *Cursors* (Linha 9).

Algoritmo 1 – Função *init*

```
1:  Parameters
2:      Cursor[] cursors
3:      Writer writer
4:      Config config                          ▷ Configurações do middleware
5:
6:  function INIT
7:      proxy ← newSocket(config.port())   ▷ Proxy é uma abstração do Socket do Gateway
8:      writer ← getWriter(config.logId())
9:      for info in config.replicasInfo() do
10:         cursor ← newCursor(info)
11:         cursors.add(cursor)
12:     end for
13:     while true do
14:         client ← proxy.accept()          ▷ Client é uma abstração do Socket do cliente
15:         handleClient(client)
16:     end while
17: end function
```

Fonte: Elaborado pela autora

Após a conexão de um cliente, a execução é direcionada para a função *handleClient*, descrita no Algoritmo 2, que é responsável por solicitar ao *Writer* que cada requisição do cliente seja registrada no *log*. O *Gateway* também repassa a referência ao método *callbackAddEntry*, o qual é invocado após a confirmação do registro no *log* (Linha 3).

Algoritmo 2 – Função *handleClient*

```
1: function HANDLECLIENT(client)          ▷ Client é uma abstração do Socket do cliente
2:     while request ← client.receive() do
3:         entryId ← writer.write(request, callbackAddEntry)
4:     end while
5: end function
```

Fonte: Elaborado pela autora

Este método, conforme exposto no Algoritmo 3, é responsável por notificar os *Cursors* disponíveis, através da função $notifyEntryAvailable(entryId)$, que o registro de $entryId$ informado foi registrado no *log* e está disponível para transmissão para as réplicas (Linha 7). Essa função disponibiliza para o *Gateway* uma $CompletableFuture$ (extensão da interface $Future$), responsável por disponibilizar artefatos para o processamento da tarefas assíncronas (Oracle Corporation, 2014). Neste contexto, a $Future$ atua como um valor imediato para o conteúdo de uma operação assíncrona, conteúdo este que estará, eventualmente, disponível.

O *Gateway* agrupa as *futures* de todos os *Cursors* notificados (Linha 8) e invoca uma função $anyOf()$ (Linha 10), responsável por aguardar pela devolutiva da requisição por algum dos *Cursors* para efetivamente encaminhá-la para o cliente (Linha 11).

Algoritmo 3 – Função *callbackAddEntry*

```
1: Parameters
2:     Cursor[] cursors
3:     Future[] futures                ▷ Future é a representação da CompletableFuture
4:
5: function CALLBACKADDENTRY(entryId, client)
6:     for cursor in cursors do
7:         future ← cursor.notifyEntryAvailable(entryId)
8:         futures.add(future)
9:     end for
10:    result ← futures.anyOf()
11:    client.send(reply)
12: end function
```

Fonte: Elaborado pela autora

O *BookKeeperWriter* (implementação do *Writer*) tem como função abstrair os métodos de registro aos *ledgers* do BookKeeper. O Algoritmo 4 exemplifica a implementação do método $write(byte[]\ data,\ Callback\ callback)$, o qual registra a sequência de *bytes* informada através da interface *WriteHandle* (Linha 5) do BookKeeper e, após o registro, executa o *callback* (Linha 6).

Algoritmo 4 – Função *write*

```
1: Parameters
2:     WriteHandle writer
3:
4: function WRITE(data, callback)
5:     entryId ← writer.append(data)
6:     callback.onComplete()
7:     return entryId
8: end function
```

Fonte: Elaborado pela autora

O *BookKeeperReader* (implementação do *Reader*), tal qual o *BookKeeperWriter*, tem com função abstrair os métodos de leitura aos *ledgers* do BookKeeper. O método $read(long\ firstEntryId, long\ lastEntryId)$, como exposto no Algoritmo 5, é responsável pela leitura dos registros entre os *IDs* informados através da interface *ReadHandle* do BookKeeper (Linha 6). Após a operação, o método executa a ação de transformar cada registro lido na abstração *Entry* do *logger* (Linha 8).

Algoritmo 5 – Função *read*

```
1: Parameters
2:     ReadHandle reader
3:     Entry[] entries
4:
5: function READ(firstEntryId, lastEntryId)
6:     ledgerEntries ← reader.read(firstEntryId, lastEntryId)
7:     for ledgerEntry in ledgerEntries do
8:         entry ← transform(ledgerEntry)
9:         entries.add(entry)
10:    end for
11:    return entries
12: end function
```

Fonte: Elaborado pela autora

O *SocketCursor* (implementação do *Cursor*) é responsável por encaminhar as requisições dos clientes para as réplicas disponíveis, em que cada *Cursor* é responsável, exclusivamente, por uma única réplica. O Algoritmo 6 ilustra essa atribuição, em que é, inicialmente, verificado se a requisição já foi previamente processada pela réplica através da verificação da variável *lastReadEntryId* com a *entryId* (Linha 8) e, caso positivo, apenas recupera o resultado do processamento a partir da consulta ao mapa *replyMap* (Linha 9).

Esta verificação é necessária pois como há um alto nível de concorrência pela função, é possível que alguma *thread* responsável por uma requisição mais avançada no *log* execute a função previamente e, consequentemente, processe todas as requisições anteriores à solicitada. Assim, o *Cursor* sempre opera sobre um conjunto de requisições, sem a necessidade de implementar estratégias para coordenar a ordenação das requisições entre as *threads* antes de transferi-las para as réplicas. Após essa verificação, a função é responsável por recuperar do *log*, através da interface *Reader*, as requisições entre *lastReadEntryId* e *entryId* (Linha 11) e encaminhar todo o conjunto de requisições para a réplica (Linha 13).

Algoritmo 6 – Função *notifyEntryAvailable*

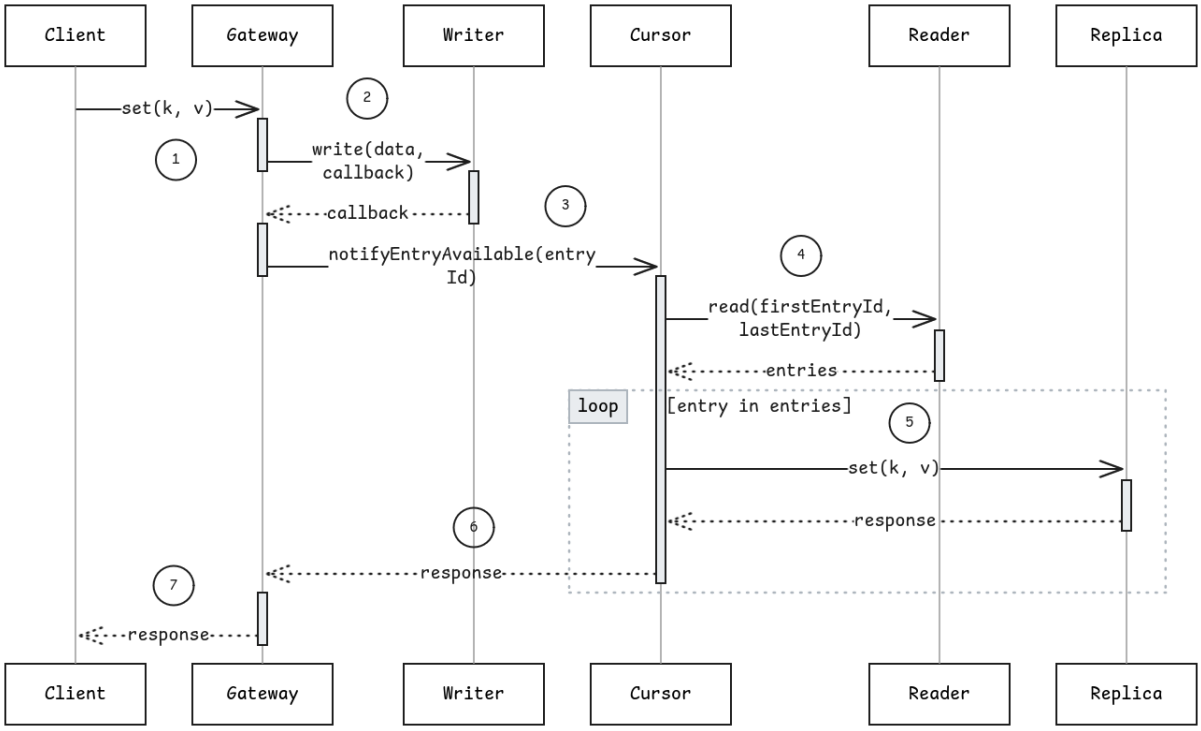
```
1:  Parameters
2:      Reader reader
3:      Replica replica                          ▷ Replica é uma abstração do Socket da réplica
4:      ID, Reply[] replyMap      ▷ Map composto pela entryId e pelo reply associado a este ID
5:      ID lastReadEntryId
6:
7:  function NOTIFYENTRYAVAILABLE(entryId)
8:      if lastReadEntryId > entryId then
9:          return replyMap.get(entryId)
10:     end if
11:     entries ← reader.read(lastReadEntryId, entryId)
12:     for entry in entries do
13:         replica.send(entry)
14:         replyMap.add(entry.getId(), replica.receive())
15:     end for
16:     lastReadEntryId ← entryId
17:     return replyMap.get(entryId)
18: end function
```

Fonte: Elaborado pela autora

O diagrama, retratado na Figura 11, ilustra a dinâmica entre o Middle-earth, uma determinada aplicação e um de seus clientes. Nele está retratado os módulos supracitados, com os respectivos métodos, e o fluxo temporal entre eles a partir de uma requisição hipotética de

um cliente. A seguir, são descritas as etapas do processo de interceptação do *logger*, dado a numeração contida no diagrama.

Figura 11 – Interação entre os módulos do *logger*



Fonte: Elaborado pela autora

1. O cliente solicita uma requisição $set(k, v)$ à aplicação, a qual é interceptada pelo *Gateway*;

2. O *Gateway* solicita ao *Writer*, por intermédio do método $write(data, callback)$, que a requisição seja registrada no *log*. Após a confirmação do registro pelo *Writer*, através da função *callback*, o *Gateway* prossegue com a notificação.

3. O *Gateway* aciona o método $notifyEntryAvailable(entryId)$, para indicar aos *Cursors* de que há uma requisição pendente para as réplicas. Após a notificação, o *Gateway* permanece ocioso até que algum *Cursor* transmita o resultado da requisição para ele.

4. O *Cursor* solicita ao *Reader* todas os registros desde o último ID transmitido para a réplica até o ID notificado através do método $read(firstEntryId, lastEntryId)$;

5. O *Cursor* itera sobre as requisições disponíveis e transmite-as para a réplica (o diagrama consta com uma única instância de *Cursor* e réplica para fins de ilustração);

6. Após o processamento da requisição pela réplica, o *Cursor* transfere o resultado para o *Gateway*;

7. O *Gateway* após o receber o resultado por qualquer um dos *Cursors*, executa a devolutiva para o cliente.

5 AVALIAÇÃO EXPERIMENTAL

Neste capítulo, são discutidos os experimentos realizados com o objetivo de mensurar os custos associados a inserção do Middle-earth em uma arquitetura *client-server*. O desempenho do Middle-earth foi avaliado com o Apache BookKeeper como *log* distribuído e uma implementação *in-memory* do *log*, com o objetivo de determinar o impacto apenas da intercepção das requisições pelo *logger*. Ademais, foi avaliada a escalabilidade do Middle-earth ao variar a quantidade de réplicas de uma aplicação e, ainda, quando outras aplicações são adicionadas a infraestrutura.

5.1 CENÁRIO DE TESTE

Toda a avaliação foi efetuada a partir de um banco de dados chave-valor em memória e seus respectivos clientes. O banco de dados foi desenvolvido para realizar a interpretação dos métodos $get(k)$, $set(k,v)$ e $delete(k)$, em que k é o inteiro aleatório e v uma *string* configurável. A avaliação, todavia, concentrou-se apenas nos métodos *get* e *set*. O gerador de carga foi desenvolvido para simular as n instâncias de clientes, em que cada cliente é responsável por solicitar requisições ao servidor por um tempo determinado. A seguir, são detalhados os parâmetros do gerador de carga.

- $clients$: Quantidade de clientes concorrentes que o gerador de carga deve simular;
- $serverHost/serverPort$: Este parâmetro define o sistema para o qual as requisições serão transferidas, especificando, portanto, o alvo do teste de carga;
- $testTime$: Ele determina a duração total em que o gerador de carga simulará as requisições dos n clientes;
- $percentRead$: Este valor estabelece a proporção de requisições de leitura em relação a outras operações que os clientes executarão ao longo do teste;
- $payload$: Refere-se ao volume de dados, em *bytes*, que é transmitido no corpo de cada requisição;
- $thinkTime$: Determina o período de tempo entre as solicitações dos clientes (*delay*). Este valor tem como objetivo simular o comportamento padrão dos clientes da aplicação.

5.2 ESTRATÉGIAS

Para a avaliação, houve a configuração de três estratégias, *client-server*, *in-memory* Middle-earth e Bk Middle-earth. As Figuras 12, 13 e 14 ilustram, respectivamente, tais estratégias. A estratégia *client-server* executou um único servidor com o banco de dados que estabelecia comunicação diretamente com os clientes. Para as estratégias com o Middle-earth configurou-se réplicas desse banco de dados, além da adição do *logger* como interceptador das requisições dos clientes. A diferença entre as duas é apenas o *log* distribuído, para a implementação *in-memory* manteve-se o *log* acoplado ao *logger* com o objetivo de determinar as implicações associadas apenas à interceptação das requisições. Enquanto para a Bk Middle-earth foi aplicado o Apache BookKeeper como o *log* distribuído.

5.3 AMBIENTE DE TESTE

Para a avaliação experimental, foi utilizado o Emulab (WHITE et al., 2002), que disponibiliza uma infraestrutura *online* e compartilhada para fins acadêmicos. O *cluster* do Emulab consta com nodos de diferentes configurações e, para os experimentos, foi selecionado o *d*710[1], cuja especificação é apresentada na Tabela 2.

Tabela 2 – Especificação dos nodos utilizados na avaliação

Especificação
Servidor Dell Poweredge R710
Processador Intel Xeon E5530 4-core
Memória RAM DDR2 de 12GB
Western Digital SATA de 500GB e Seagate SATA de 250GB
Sistema Operacional Ubuntu 22.04
Java 17

Fonte: Elaborado pela autora

Os parâmetros utilizados no gerador de carga estão descritos na Tabela 3, na qual estão detalhados o tempo de execução do teste, o *think-time* entre as requisições, o percentual de solicitações de leitura, a quantidade de *bytes* transmitidos e, por fim, a quantidade de clientes para cada experimento. O tempo de execução foi configurado em 2 minutos para garantir uma amostragem significativa dos dados de vazão e latência. Configurou-se o *think-time* em 1 milissegundo (ms) para permitir uma saturação rápida das aplicações. Para as operações, foi definida

[1] https://gitlab.flux.utah.edu/emulab/emulab-devel/-/wikis/Utah-Cluster/d710s

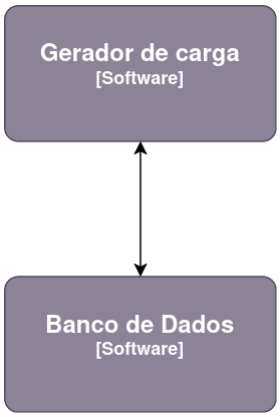


Figura 12 – Estratégia *client-server*

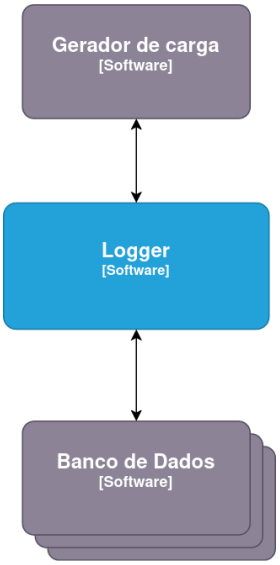


Figura 13 – Estratégia *in-memory* Middle-earth

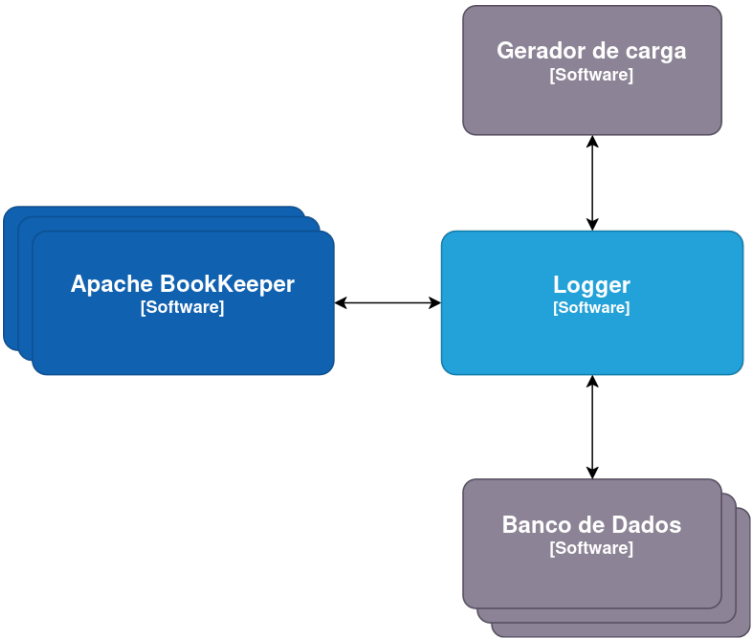


Figura 14 – Estratégia Bk Middle-earth

uma proporção uniforme, assim 50% das operações são de leitura $get(k)$ e o remanescente de escrita $set(k,v)$. O *payload* selecionado foi de 1024 bytes (1 kB) para avaliar os cenários com um valor usual em APIs RESTful, porém antes da fragmentação dos pacotes. Outro parâmetro informado no gerador de carga é a quantidade de clientes que ele deve simular. Inicialmente, é definida uma carga de n clientes concorrentes e essa carga é gradualmente incrementada até a identificação da saturação. Cada estágio de carga foi submetido a duas repetições independentes para assegurar que os valores coletados não fossem reflexo de uma oscilação momentânea do ambiente de teste.

Tabela 3 – Parâmetros utilizados no gerador de carga

Parâmetro	**Valor**
Tempo de execução	2 minutos
Think-time	1 milissegundo (ms)
Percentual de leitura	50%
Payload	1024 bytes (1 kB)
Clientes	Gradual (e.g. 2, 4, 8, etc)

Fonte: Elaborado pela autora

A vazão para cada variação do experimento foi calculada a partir da média da vazão coletada a cada segundo de execução do banco de dados, excluindo-se os valores nulos, que correspondem aos instantes ociosos do servidor. Para a latência, foi selecionada a média e o 95^o percentil dos tempos entre a solicitação e a devolutiva de uma requisição. A média demonstra o comportamento padrão dos clientes, já o 95^o percentil apresenta o cenário com elevada latência, porém desconsidera *outliers*.

5.4 EXPERIMENTOS

O foco dos experimentos foi mensurar o desempenho do Middle-earth em contraste com a implementação *client-server*, a partir da análise da vazão e da latência. Outro objetivo foi avaliar a escalabilidade do *logger* e do *log* distribuído ao variar, respectivamente, o número de réplicas de uma aplicação e o número de aplicações associadas ao Middle-earth.

A distribuição dos nodos alocados para os experimentos está ilustrada na Tabela 4. Para a avaliação *client-server* configurou-se um nodo para atuar como o banco de dados chave-valor e outro para executar o gerador de carga. Para a estratégia *in-memory* Middle-earth, além do banco de dados e do gerador de carga, foi configurada uma instância do *logger* entre a aplicação e os clientes. Já para a implementação Bk Middle-earth, a implementação em memória do *log* é substituída pelo Apache BookKeeper, exigindo a instância de três *bookies* e uma do ZooKeeper. Visando mensurar a escalabilidade do *logger*, as estratégias *in-memory* e Bk têm duas variações, com uma e duas réplicas do banco de dados.

A vazão máxima das estratégias mencionadas foi calculada dado a média da vazão no ponto de saturação de cada estratégia, as quais estão ilustradas na Figura 15. No gráfico, é possível visualizar que, enquanto a *client-server* obteve uma vazão máxima próxima de 80.000 comandos/s, as estratégias com o Middle-earth estão apenas entre os 3000 comandos/s. A princípio, associa-se essa questão a restrição das aplicações a uma execução *single-thread* nas estra-

Tabela 4 – Distribuição dos nodos para a avaliação do Middle-earth

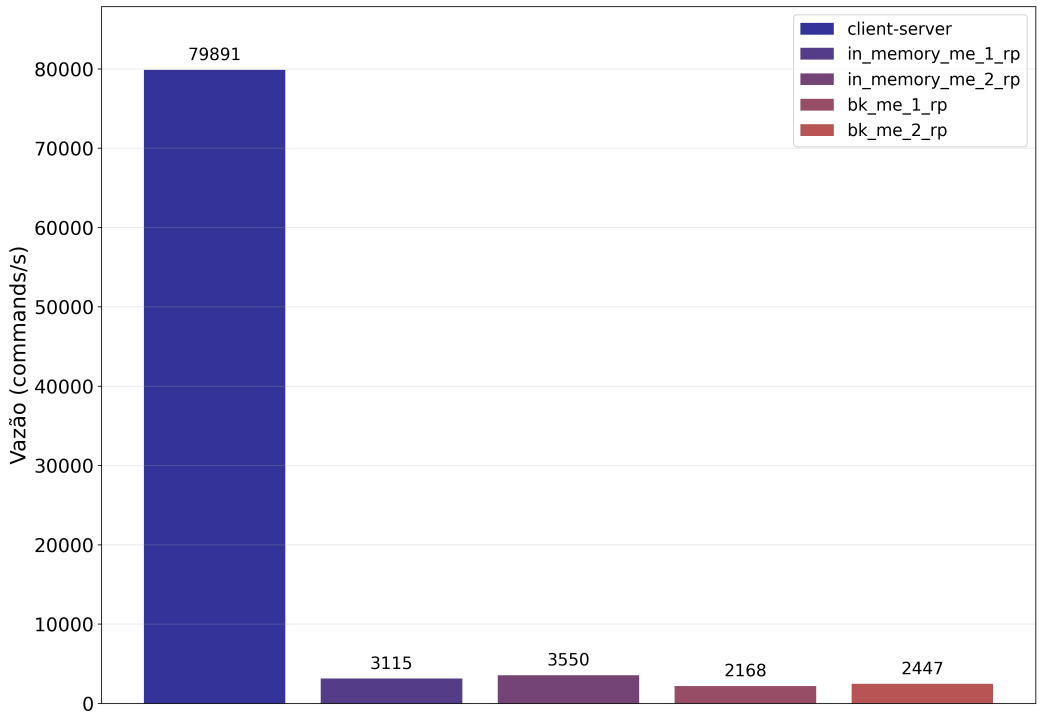
Variações	Cliente	Banco de dados	Logger	Zookeeper	BookKeeper
client-server	1	1	–	–	–
in_memory_me_1_rp	1	1	1	–	–
in_memory_me_2_rp	1	2	1	–	–
bk_me_1_rp	1	1	1	1	3
bk_me_2_rp	1	2	1	1	3

Fonte: Elaborado pela autora

tégias com o Middle-earth, para respeitar os princípios da Replicação de Máquinas de Estado (RME). Na Seção 6.1 são discutidas estratégias para contornar essa limitação, porém, para o escopo desta monografia, essa restrição foi mantida.

Como houve uma discrepância na vazão das estratégias, não foi possível fixar todas elas em um único gráfico. Em vista disso, os gráficos a seguir irão ilustrar o desempenho da estratégia *client-server* em comparação a *in-memory* Middle-earth e, posteriormente, da *in-memory* Middle-earth com a Bk Middle-earth.

Figura 15 – Vazão máxima das estratégias utilizadas na avaliação



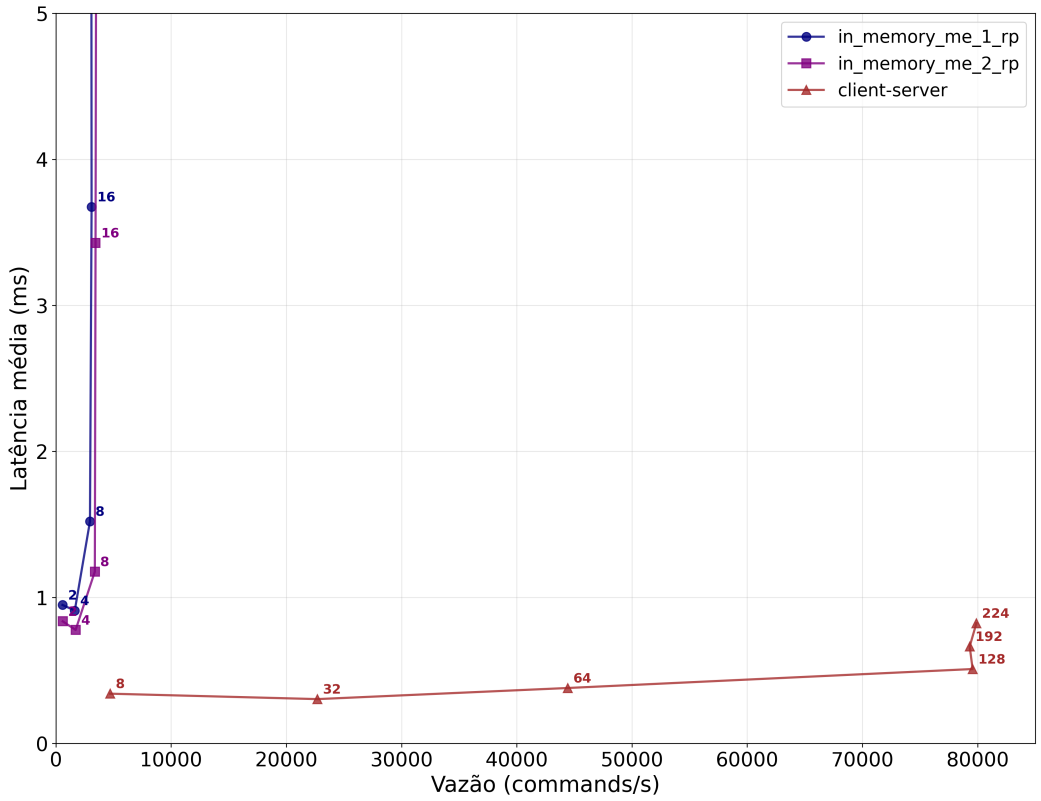
Fonte: Elaborado pela autora

Cada variação do experimento iniciou com uma determinada carga de clientes concorrentes, sendo esta carga gradualmente incrementada até a sua saturação. Assim, a vazão *versus*

à latência média e 95^{o} percentil das estratégias *client-server* e *in-memory* Middle-earth são ilustradas, respectivamente, na Figura 16 e na Figura 17. Nestes gráficos, é notável que o ponto de saturação da estratégia *client-server* é próximo de 80.000 comandos/s, enquanto a *in-memory* Middle-earth, em suas duas variações, obteve seu ponto de saturação com apenas 4.000 comandos/s. Os valores de latência, porém, não são muito afetados, evidenciando que o custo do Middle-earth reside na degradação da vazão, já que a variação da latência é uma consequência natural da adição de um serviço de interceptação.

Contudo, até a saturação do *in-memory* Middle-earth, as duas estratégias têm uma vazão equivalente dado o mesmo número de clientes concorrentes, demonstrando que o Middle-earth conseguiu transferir as requisições em uma vazão semelhante até este ponto. Cenário que indica que a limitação da vazão se deve, justamente, a restrição das aplicações à execução *single-thread*, em que a vazão do *middleware* é limitada ao máximo da vazão de uma *thread* com carga máxima.

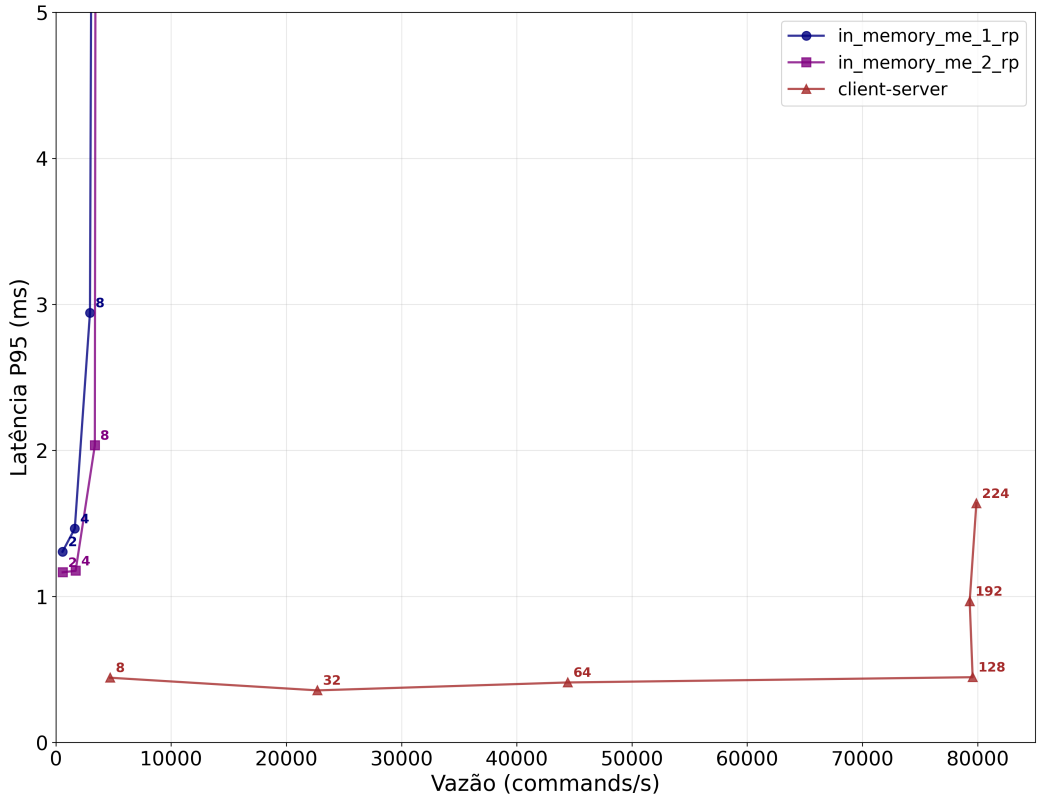
Figura 16 – Latência client-server vs *in-memory* Middle-earth (média)



Fonte: Elaborado pela autora

A avaliação da estratégia *in-memory* Middle-earth, em contraste com a Bk Middle-earth, focou na latência e vazão destas à medida que o número de clientes foi, gradualmente, incrementado até a saturação do banco de dados. Neste contexto, a Figura 18 demonstra a latência média, enquanto a Figura 19 apresenta o 95^{o} percentil da latência para esta configuração.

Figura 17 – Latência client-server vs *in-memory* Middle-earth (95^o percentil)



Fonte: Elaborado pela autora

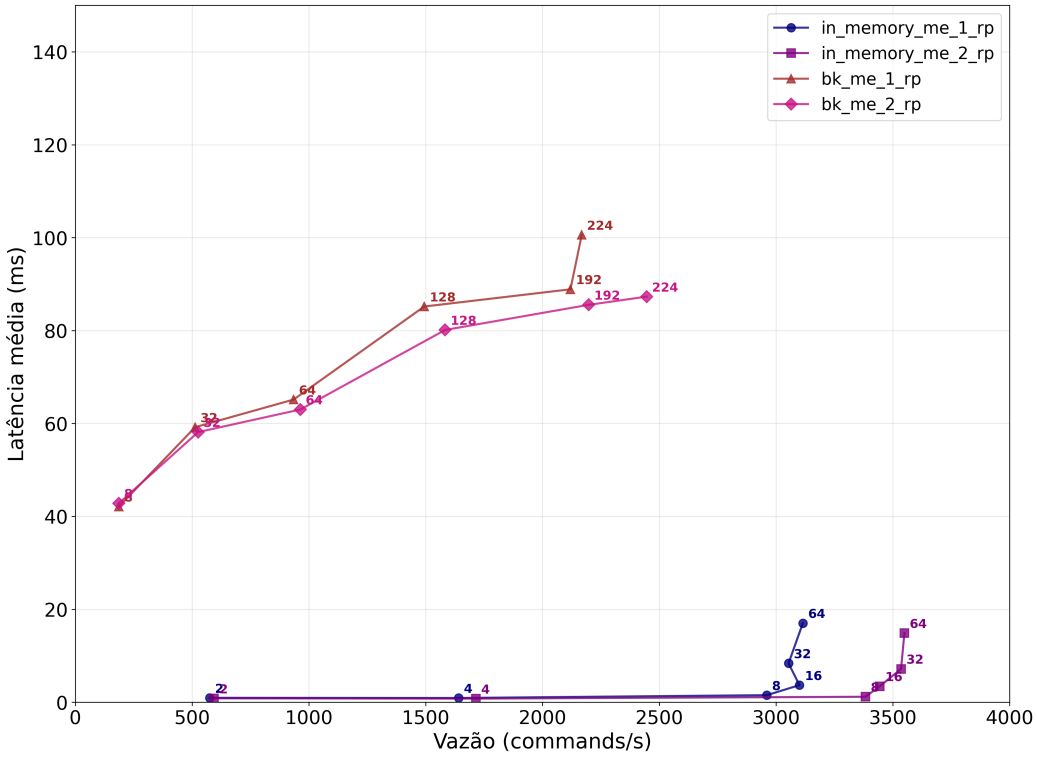
Nos gráficos, é notável que a estratégia com o BookKeeper necessitou de uma quantidade de clientes superior à da estratégia *in-memory* para atingir determinados pontos de vazão. Enquanto a estratégia *in-memory* atingiu uma vazão de aproximadamente 500 comandos/s com apenas 2 clientes, o BookKeeper necessitou de 32 clientes para atingir essa mesma vazão. Além do fato de que o Middle-earth, com o BookKeeper, não atingiu realmente um ponto de saturação. Todavia, essas questões são possivelmente uma consequência da adição do *log* distribuído. Isso é atribuído às estratégias de sincronização adotadas pelos *bookies* e pelo ZooKeeper, as quais mitigam a manifestação da saturação, já que o *logger* fica, obrigatoriamente, ocioso pelo período de comunicação entre os serviços.

5.4.1 Escalabilidade do *logger*

Nota-se nos gráficos discutidos que, apesar de o desempenho do Middle-earth ser inferior à estratégia *client-server*, o *logger* manteve-se estável após a adição de outras réplicas. Essa estabilidade foi observada tanto para a estratégia *in-memory* quanto para a estratégia com o BookKeeper. Evidenciando a escalabilidade do *logger* quando outras réplicas são associadas ao Middle-earth.

O desempenho do *logger* não é afetado pela adição de réplicas, pois o volume de réplicas não possui interferência direta no processo de durabilidade dos dados e coordenação das requisições. O Middle-earth foi projetado para processar a devolutiva ao cliente após o processamento por qualquer réplica, independentemente da quantidade de réplicas. Essa estratégia é refletida nos experimentos, nos quais o desempenho das variações com duas réplicas foi ligeiramente superior ao desempenho das variações com uma única réplica.

Figura 18 – Latência *in-memory* Middle-earth vs Bk Middle-earth (média)



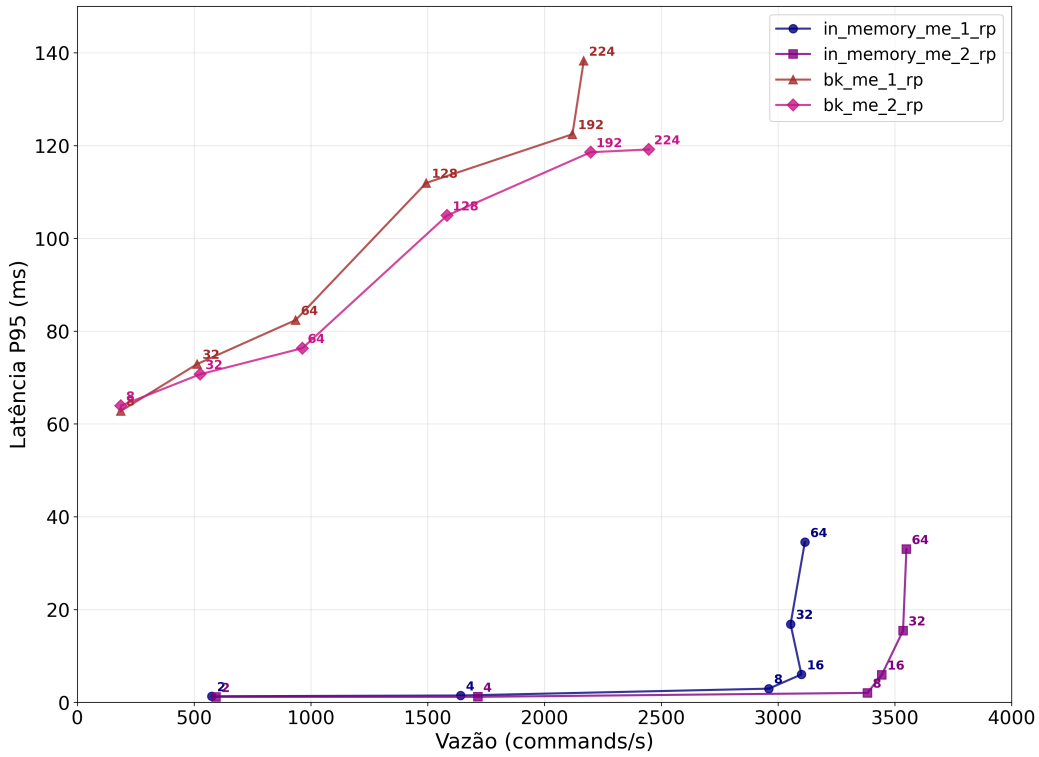
Fonte: Elaborado pela autora

5.4.2 Escalabilidade do *log* distribuído

Outro objetivo da avaliação experimental é mensurar a escalabilidade do *log* distribuído, mais especificamente, do Apache BookKeeper, quando aplicações distintas são adicionadas ao Middle-earth. A Tabela 5 ilustra a distribuição dos nodos para essa configuração. Configurou-se dois bancos de dados chave-valor distintos, com uma única réplica. Consequentemente, foi estabelecido um gerador de carga para cada aplicação. Duas instâncias do *logger*, em que cada *logger* é responsável por uma única aplicação. A distribuição de nodos para o BookKeeper manteve-se igual, com três *bookies* e uma instância do ZooKeeper.

O foco desta avaliação, tal como nos demais experimentos, foi a análise comparativa entre a vazão e a latência das duas variações. Os resultados de cada variação são representados

Figura 19 – Latência *in-memory* Middle-earth vs Bk Middle-earth (95^o percentil)



Fonte: Elaborado pela autora

Tabela 5 – Distribuição dos nodos na avaliação com aplicações compartilhando o *log*

Variações	Cliente	Banco de dados	Logger	Zookeeper	BookKeeper
bk_me_1_app	1	1	1	1	3
bk_me_2_app	2	2	2	1	3

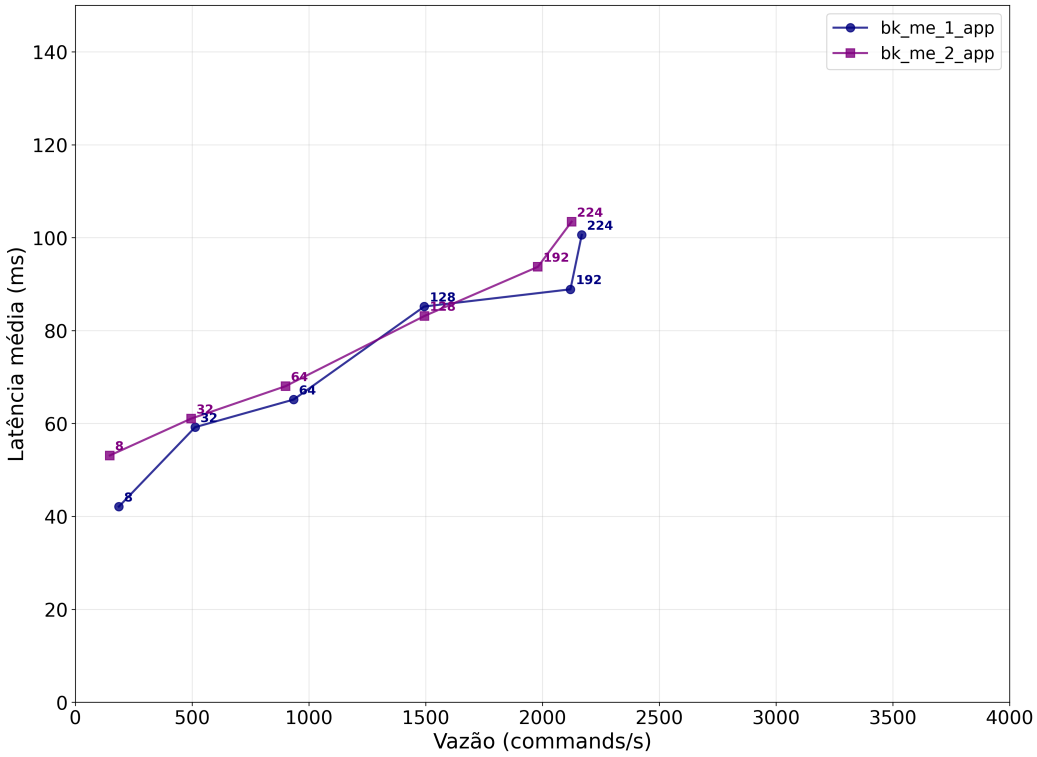
Fonte: Elaborado pela autora

para um valor idêntico de clientes concorrentes. Assim, na Figura 20 e na Figura 21 estão ilustradas, respectivamente, a vazão *versus* a latência média e a vazão *versus* a latência 95^o percentil do Bk Middle-earth para uma e duas aplicações compartilhando o *log* distribuído. Nos gráficos, é notável que a vazão e a latência nas duas configurações são equivalentes, fato que corrobora o potencial de escalabilidade do *log* distribuído dentro do Middle-earth quando aplicações distintas são adicionadas à infraestrutura.

Nos gráficos, nota-se que o custo do compartilhamento do *log* distribuído reside em um pequeno acréscimo da latência. Isso é evidenciado pela latência máxima de 100.59ms registrada pela variação bk_me_1_app, enquanto a bk_me_2_app alcançou 103.42ms. Todavia, esse custo é inerente e, normalmente, constitui um valor aceitável em sistemas distribuídos em prol da disponibilidade e confiabilidade.

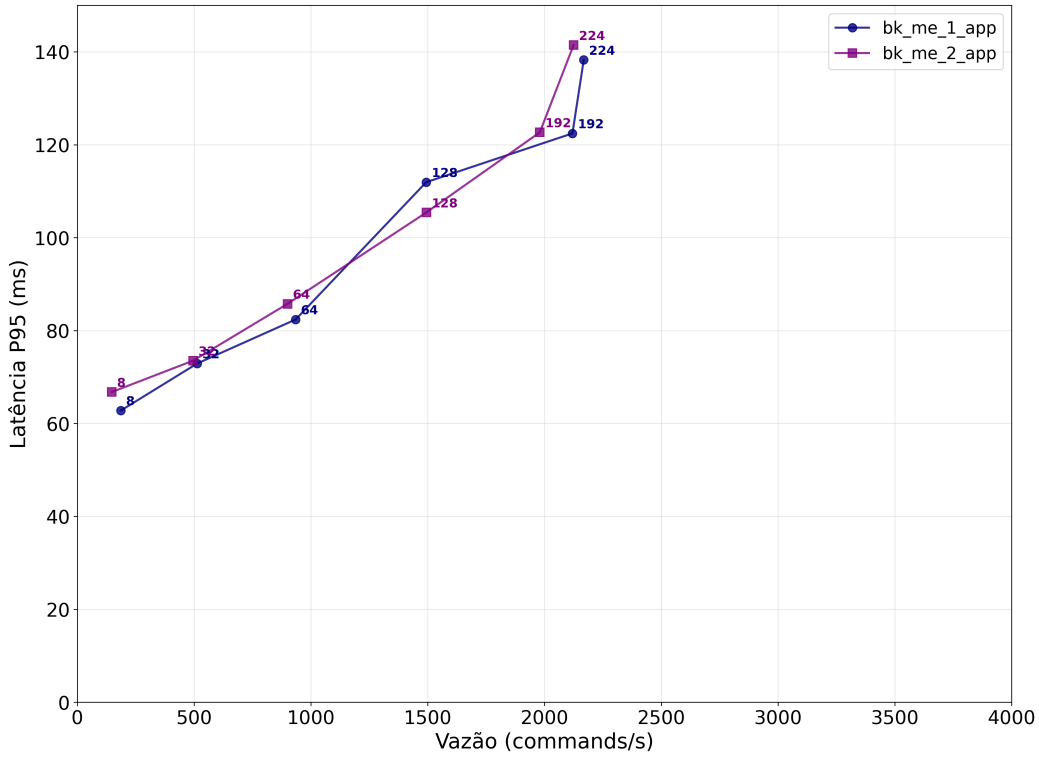
Neste contexto, apesar de o Middle-earth, ao empregar o BookKeeper como *log* distribuído, apresentar lacunas no desempenho quando comparado com as outras estratégias, permite o compartilhamento de recursos ao compartilhar a mesma instância de *log* distribuído. Por conseguinte, essa estratégia viabiliza a alocação de recursos de modo direcionado dentro do Middle-earth, possibilitando operar as réplicas da aplicação e o próprio *logger* a partir de estratégias *serverless*.

Figura 20 – Latência Bk Middle-earth com duas aplicações alvo (média)



Fonte: Elaborado pela autora

Figura 21 – Latência Bk Middle-earth com duas aplicações alvo (95^o percentil)



Fonte: Elaborado pela autora

6 CONCLUSÃO

Esta monografia explorou o desenvolvimento de um serviço de replicação como *middleware*, com o objetivo de proporcionar alta modularidade, independência e escalabilidade ao implementar um serviço que desloca a estratégia de replicação das réplicas de uma aplicação, possibilitando que as réplicas atuem apenas no processamento de solicitações. Ao deslocar tal responsabilidade das réplicas, o Middle-earth visa, ainda, proporcionar aos projetistas otimizações na implementação da replicação, já que o serviço é independente da aplicação.

A monografia, para construir sua fundamentação teórica, discorreu quanto as estratégias de replicação comumente aplicadas para assegurar a disponibilidade dos serviços: a replicação passiva (ou *primary-backup*) e a replicação ativa (ou Replicação de Máquinas de Estados). Ainda neste tópico, apresentou-se dois serviços adotados no desenvolvimento do Middle-earth, o Apache BookKeeper, como *log* distribuído da arquitetura, e o Apache ZooKeeper, um serviço de coordenação distribuído, para a gerência dos metadados do *middleware*.

Após essa contextualização, a monografia descreveu a concepção do Middle-earth e a implementação de uma POC dele. O Middle-earth é composto por dois serviços independentes, um log distribuído e um denominado *logger*. Enquanto o *log* distribuído é responsável por assegurar a durabilidade e a disponibilidade dos dados, o *logger* é responsável por interceptar as requisições de uma determinada aplicação, delegar o registro delas ao *log* distribuído e coordenar a comunicação entre as réplicas dessa aplicação. Para a implementação da POC, foi selecionado o Apache BookKeeper, associado ao Apache ZooKeeper, como *log* distribuído e adotou-se uma comunicação via *Sockets*, devido a facilidade de implementação do mecanismo.

O Middle-earth foi submetido a uma avaliação experimental no Emulab, plataforma que disponibiliza uma infraestrutura *online* e compartilhada, com o objetivo de mensurar sua vazão, latência e escalabilidade em contraposição a uma estratégia sem o *middleware*. Visando determinar as implicações associadas apenas à interceptação, o Middle-earth foi também avaliado em duas configurações, uma que empregou o BookKeeper como *log* distribuído e outra que manteve o *log* acoplado ao *logger*.

A partir dos resultados obtidos, foi possível visualizar que, apesar do desempenho do Middle-earth ser inferior a estratégia sem implementação de replicação, ele demonstrou-se altamente modular, escalável e, principalmente, independente da aplicação alvo. Fato que possibilita que ele seja facilmente integrado na arquitetura de uma aplicação e proporcione otimizações no desenvolvimento. Ilustrando que é uma proposta promissora dentro da replicação e que possui espaço para avaliações a fim de investigar possíveis correções em seu desempenho.

6.1 TRABALHOS FUTUROS

Alguns trabalhos futuros são a avaliação de possíveis otimizações no desempenho do Middle-earth, dada a discrepância em relação à implementação sem replicação. Naturalmente, o serviço irá apresentar uma vazão inferior, já que as réplicas da aplicação deverão ser *single-thread*, com o objetivo de respeitar uma restrição da própria Replicação de Máquinas de Estados. Todavia, a avaliação de estratégias de comunicação em lotes com as réplicas é uma alternativa para suavizar essa restrição. Ademais, a avaliação da Replicação de Máquinas de Estados Paralela para o Middle-earth é uma proposta promissora para elevar o desempenho do *middleware*, apesar da exigência de conhecimento da semântica do serviço replicado.

Atualmente, devido a limitação do Apache BookKeeper de um único *writer* por *log*, o *logger* é um ponto único de falha dentro da arquitetura, já que não é possível instanciar réplicas do *logger* para operar em conjunto com o *log* distribuído. Nessa perspectiva, outro trabalho futuro é a avaliação da possibilidade de utilizar outro serviço para o *log* distribuído ou construir um *log* distribuído próprio a partir de estratégias de replicação.

O Middle-earth foi desenvolvido a partir de uma comunicação via *sockets* para construir uma interface de comunicação de baixo nível, ideal para validação da arquitetura do *middleware*. Para trabalhos futuros, é possível desenvolver *wrappers* para outros padrões de comunicação, a fim de permitir a integração com protocolos e *frameworks* amplamente utilizados na indústria. O desenvolvimento desses *wrappers* permitirá avaliar a aplicabilidade do Middle-earth em serviços já utilizados pelas organizações e avaliar seu desempenho em outros contextos, indo além da comunicação direta via *sockets*.

REFERÊNCIAS

ALCHIERI, E. et al. Boosting state machine replication with concurrent execution. In: **2018 Eighth Latin-American Symposium on Dependable Computing (LADC)**. [S.l.: s.n.], 2018. p. 77–86.

BACON, D. F. et al. Spanner: Becoming a sql system. In: **Proceedings of the 2017 ACM International Conference on Management of Data**. New York, NY, USA: Association for Computing Machinery, 2017. (SIGMOD '17), p. 331–343. ISBN 9781450341974. Disponível em: https://doi.org/10.1145/3035918.3056103.

BAKER, J. et al. Megastore: Providing scalable, highly available storage for interactive services. In: **Proceedings of the Conference on Innovative Data system Research (CIDR)**. [s.n.], 2011. p. 223–234. Disponível em: http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper32.pdf.

BALAKRISHNAN, M. et al. Corfu: A distributed shared log. **ACM Trans. Comput. Syst.**, Association for Computing Machinery, New York, NY, USA, v. 31, n. 4, dez. 2013. ISSN 0734-2071. Disponível em: https://doi.org/10.1145/2535930.

BBC News Brasil. **Canva, Snapchat, Roblox: por que aplicativos de centenas de empresas ficaram fora do ar, segundo serviço de nuvem da Amazon**. 2025. Disponível em: https://www.bbc.com/portuguese/articles/cgql04yw9q0o.

BUDHIRAJA, N. et al. The primary-backup approach. In: ____. **Distributed Systems (2nd Ed.)**. USA: ACM Press/Addison-Wesley Publishing Co., 1993. p. 199–216. ISBN 0201624273.

COELHO, F. et al. Loom: A closed-box disaggregated database system. In: **Proceedings of the 12th Latin-American Symposium on Dependable and Secure Computing**. New York, NY, USA: Association for Computing Machinery, 2023. (LADC '23), p. 30–39. ISBN 9798400708442. Disponível em: https://doi.org/10.1145/3615366.3615424.

CORBETT, J. C. et al. Spanner: Google's globally distributed database. **ACM Trans. Comput. Syst.**, Association for Computing Machinery, New York, NY, USA, v. 31, n. 3, ago. 2013. ISSN 0734-2071. Disponível em: https://doi.org/10.1145/2491245.

CRISTIAN, F. et al. Atomic broadcast: From simple message diffusion to byzantine agreement. **Information and Computation**, v. 118, n. 1, p. 158–179, 1995. ISSN 0890-5401. Disponível em: https://www.sciencedirect.com/science/article/pii/S0890540185710607.

GRACIA-TINEDO, R. et al. Pravega: A tiered storage system for data streams. In: **Proceedings of the 24th International Middleware Conference**. New York, NY, USA: Association for Computing Machinery, 2023. (Middleware '23), p. 165–177. ISBN 9798400701771. Disponível em: https://doi.org/10.1145/3590140.3629113.

GUO, S.; DHAMANKAR, R.; STEWART, L. Distributedlog: A high performance replicated log service. In: **2017 IEEE 33rd International Conference on Data Engineering (ICDE)**. [S.l.: s.n.], 2017. p. 1183–1194.

HALOI, S. **Apache ZooKeeper Essentials**. [S.l.]: Packt Publishing, 2015. ISBN 1784391328.

JUNQUEIRA, F.; REED, B. **ZooKeeper: Distributed Process Coordination**. 1st. ed. [S.l.]: O'Reilly Media, Inc., 2013. ISBN 1449361307.

JUNQUEIRA, F. P.; KELLY, I.; REED, B. Durability with bookkeeper. Association for Computing Machinery, New York, NY, USA, v. 47, n. 1, p. 9–15, jan 2013. ISSN 0163-5980. Disponível em: https://doi.org/10.1145/2433140.2433144.

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

LAMPORT, L. The part-time parliament. **ACM Trans. Comput. Syst.**, Association for Computing Machinery, New York, NY, USA, v. 16, n. 2, p. 133–169, may 1998. ISSN 0734-2071. Disponível em: https://doi.org/10.1145/279227.279229.

MARANDI, P. J.; BEZERRA, C. E.; PEDONE, F. Rethinking state-machine replication for parallelism. In: **Proceedings of the 2014 IEEE 34th International Conference on Distributed Computing Systems**. USA: IEEE Computer Society, 2014. (ICDCS '14), p. 368–377. ISBN 9781479951697. Disponível em: https://doi.org/10.1109/ICDCS.2014.45.

MENDIZABAL, O. M.; DOTTI, F. L.; PEDONE, F. High performance recovery for parallel state machine replication. In: **2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)**. [S.l.: s.n.], 2017. p. 34–44.

NEU, W. A. et al. **An Introduction to Cloud Databases: A Guide for Administrators**. [S.l.]: O'Reilly Media, 2019.

ONGARO, D.; OUSTERHOUT, J. In search of an understandable consensus algorithm. In: **Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference**. USA: USENIX Association, 2014. (USENIX ATC'14), p. 305–320. ISBN 9781931971102.

Oracle Corporation. **Class CompletableFuture<T> (Java Platform SE 8)**. Oracle Help Center, 2014. Disponível em: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html.

Oracle Corporation. **Java Documentation**. Oracle Help Center, 2025. Disponível em: https://docs.oracle.com/en/java/.

PENG, D.; DABEK, F. Large-scale incremental processing using distributed transactions and notifications. In: **Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation**. USA: USENIX Association, 2010. (OSDI'10), p. 251–264.

SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: A tutorial. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 22, n. 4, p. 299–319, dec 1990. ISSN 0360-0300. Disponível em: https://doi.org/10.1145/98163.98167.

SCHNEIDER, F. B. Replication management using the state-machine approach. In: ____. **Distributed Systems (2nd Ed.)**. USA: ACM Press/Addison-Wesley Publishing Co., 1993. p. 169–197. ISBN 0201624273.

SHARMA, R.; ATYAB, M. Introduction to apache pulsar. In: ____. **Cloud-Native Microservices with Apache Pulsar: Build Distributed Messaging Microservices**. Berkeley, CA: Apress, 2022. p. 1–22. ISBN 978-1-4842-7839-0. Disponível em: https://doi.org/10.1007/978-1-4842-7839-0_1.

The Apache Software Foundation. **Apache BookKeeper, Release 4.17.1**. 2024. GitHub. Disponível em: https://github.com/apache/bookkeeper/releases/tag/release-4.17.1.

The Apache Software Foundation. **Apache ZooKeeper, Release 3.8.4**. 2024. GitHub. Disponível em: https://github.com/apache/zookeeper/releases/tag/release-3.8.4.

The Apache Software Foundation. **Apache BookKeeper**. 2025. Disponível em: https://bookkeeper.apache.org.

The Apache Software Foundation. **Apache ZooKeeper**. 2025. Disponível em: https://zookeeper.apache.org/.

The Independent. **Amazon down: Internet outage hits sales**. 2021. Disponível em: https://www.independent.co.uk/news/business/amazon-down-internet-outage-sales-b1861737.html.

VERBITSKI, A. et al. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In: **Proceedings of the 2017 ACM International Conference on Management of Data**. New York, NY, USA: Association for Computing Machinery, 2017. (SIGMOD '17), p. 1041–1052. ISBN 9781450341974. Disponível em: https://doi.org/10.1145/3035918.3056101.

WHITE, B. et al. An integrated experimental environment for distributed systems and networks. In: USENIX Association. **Proc. of the Fifth Symposium on Operating Systems Design and Implementation**. Boston, MA, 2002. p. 255–270.

WHITE, T. **Hadoop: The definitive guide**. [S.l.]: "O'Reilly Media, Inc.", 2012.

XAVIER, L. G. et al. Scalable and decoupled logging for state machine replication. In: **Anais do XXXVIII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos**. [s.n.], 2020. p. 267–280. ISSN 2177-9384. Disponível em: https://sol.sbc.org.br/index.php/sbrc/article/view/12288.

APÊNDICE A – IMPLEMENTAÇÃO DO MIDDLE-EARTH

Código da implementação do Middle-earth e as aplicações desenvolvidas para os experimentos: https://github.com/rafatillmann/middle-earth

APÊNDICE B – *SCRIPTS* DESENVOLVIDOS PARA A AVALIAÇÃO EXPERIMENTAL

Os *scripts* desenvolvidos para a execução dos experimentos: https://github.com/rafatillmann/middle-earth-scripts

ANEXO A – ARTIGO ERAD

Artigo desenvolvido no período de proposta desta monografia e publicado na XXIV Escola Regional de Alto Desempenho da Região Sul (ERAD).

Proposta de um serviço de replicação desacoplado da aplicação

Rafaela Tillmann[1], **Odorico M. Mendizabal**[1]

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

rafaela.tillmann@grad.ufsc.br, odorico.mendizabal@ufsc.br

Resumo. *Na sociedade contemporânea a indisponibilidade de um serviço pode acarretar em vários prejuízos. Para assegurar a disponibilidade das aplicações, é possível replicá-las em múltiplos servidores dado dois enfoques diferentes, replicação passiva e Replicação de Máquina de Estados. Entretanto, devido aos desafios associados à implementação dessas duas estratégias, este trabalho visa propor o desenvolvimento de um serviço de replicação desacoplado.*

1. Introdução

Informações são o cerne da sociedade, conexão digital é a representação humana da contemporaneidade. Alguns segundos de inatividade em sistemas de armazenamento ou processamento de dados pode representar uma perda significativa para as organizações, como foi como para a Amazon em 2021, quando a indisponibilidade de apenas 59 minutos acarretou em uma perda de 34 milhões de dólares em vendas [Independent 2023]. Portanto, para assegurar a disponibilidade dos serviços, é importante implementar alguma estratégia de replicação dos serviços, como a replicação passiva (*primary-backup*) ou a Replicação de Máquina de Estados (SMR, do inglês *State Machine Replication*).

A replicação passiva mantém apenas um servidor para lidar com as requisições dos clientes e, após o processamento da requisição, este servidor (primário) replica o estado atualizado para os demais servidores (*backups*). Esse enfoque, apesar de assegurar a confiabilidade, pode elevar a latência das requisições e ocasionar indisponibilidade momentânea do serviço no caso de substituição do primário.

Na Replicação de Máquina de Estados, é previsto que todas as réplicas iniciem no mesmo estado, executem as mesmas requisições de modo determinístico e em uma mesma ordem, avançando por estados idênticos. Diferentemente da replicação passiva, todas as réplicas irão executar as requisições dos clientes e, antes de efetuar a operação, há necessidade de estabelecer um acordo sobre a ordenação das requisições entre as réplicas. Esse acordo é assegurado por protocolos de consenso ou protocolos de difusão atômica, os quais são estruturados em nível de aplicação, fato que eleva complexidade de implementação, visto que o projetista deve possuir conhecimento dessas técnicas.

Este trabalho, portanto, propõe o desenvolvimento de um serviço de replicação desacoplado das aplicações, que assegure a ordenação das requisições e atue como intermediário entre o cliente e as réplicas. Espera-se alcançar alta modularidade, escalabilidade e otimizações no desenvolvimento, visto que a lógica da aplicação será desassociada das réplicas.

2. Replicação

Estratégias de replicação são comumente utilizadas para prover tolerância a falhas, garantindo a execução do serviço mesmo na ocorrência de falhas em um número limitado de

réplicas. A seguir são ilustradas duas técnicas amplamente utilizadas.

Replicação passiva (ou primary-backup): é uma estratégia que foca em um único servidor responsável pelo processamento das requisições e por envio das respostas para os clientes. Após o servidor processar a solicitação do cliente, ele é responsável por transmitir o estado final para as demais réplicas, que irão atuar como *backups* caso o servidor principal seja acometido com alguma falha e necessite ser substituído [Budhiraja et al. 1993]. Essa estratégia, apesar de assegurar replicação do sistema, eleva a latência das requisições, visto que após o processamento da requisição o servidor primário deve replicar o estado para um quórum de servidores *backups* antes de efetivamente responder ao cliente. Apesar da possibilidade de responder o cliente logo após o processamento da requisição, essa ação não assegura consistência forte, já que há possibilidade do servidor ser substituído por uma réplica em um estado anterior, no caso de falhas.

Replicação de Máquina de Estados (ou State Machine Replication): é uma estratégia que prevê que todas as réplicas irão executar as requisições de clientes. Schneider aponta que se forem executadas operações determinísticas, ordenadamente, em diferentes réplicas, elas irão evoluir para estados idênticos. Portanto, o enfoque dessa estratégia é iniciar todas as réplicas em um mesmo estado e assegurar que todas as requisições serão executadas, ordenadamente, por todas as réplicas [Schneider 1990].

Para prover essa ordenação, são comumente aplicados protocolos de consenso, como o Paxos e o Raft, porém, há necessidade de implementação desses protocolos em nível de aplicação, que eleva os custos de escalabilidade da aplicação e impossibilita aplicar a solução de replicação para outros serviços, além da própria complexidade associada à implementação dos protocolos.

3. Arquitetura

O projeto visa a implementação de um serviço de replicação totalmente desacoplado, que atue como mediador entre as réplicas e os clientes, assegurando a ordenação das solicitações dos clientes. Nessa perspectiva, a arquitetura proposta visa o desenvolvimento dos aspectos listados.

3.1. Desacoplamento

Estratégias de desacoplamento de funcionalidades de serviços tolerantes a falhas não são inéditas. Como, o *DistributedLog* que é um serviço de *logs* replicados desenvolvido para auxiliar na replicação do Twitter Manhattan. Nele, o *DistributedLog* é responsável por assegurar a ordenação das requisições que serão posteriormente lidas pelas réplicas. Entretanto, ainda há necessidade de uma réplica atuar como líder, intervindo nas requisições dos clientes e na gerência das demais réplicas [Guo et al. 2017]. Xavier *et al.* e Scharf *et al.* também propõem o desenvolvimento de serviços de *logs* desacoplados, porém, as réplicas são responsáveis por estabelecer o consenso e apenas após essa ação que a sequência de *logs* é registrada [Xavier et al. 2020, Scharf et al. 2023].

Esses serviços ilustram a possibilidade de desacoplar algumas funcionalidades das réplicas, entretanto, ainda há a necessidade da atuação das réplicas no fluxo da replicação. Nessa perspectiva, o projeto prevê o desacoplamento integral, possibilitando que as réplicas atuem apenas no processamento das operações solicitadas pelos clientes, além de proporcionar para os projetistas otimizações na implementação da replicação.

3.2. Generalização

Devido à complexidade envolvida na implementação de replicação, há o desenvolvimento de inúmeras aplicações com garantia de replicação que têm possibilidade de serem incorporadas a outros serviços, como o Amazon Aurora [Verbitski et al. 2017], um SGBD com replicação automática compatível com MySQL. Entretanto, como o Amazon Aurora é construído apoiado em um *fork* do código do MySQL, ele é complexo quanto à manutenção, além de possuir uma forte dependência do MySQL.

Para contornar esse vínculo direto com o MySQL, Coelho *et al.* propõe o Loom, que usa o servidor original para facilitar implementação da replicação, além de proporcionar generalização quanto ao SGBD aplicado. Ele cria uma interface entre o banco de dados e a aplicação, possibilitando que toda estratégia de replicação seja desassociada do servidor [Coelho et al. 2023]. Entretanto, apesar de possibilitar generalização quanto ao contexto dos SGBDs, o Loom, tal como Amazon Aurora e outras soluções, está focado em um único contexto e não proporciona um serviço totalmente genérico para replicação. Nesta perspectiva, a arquitetura proposta será implementada com objetivo de possibilitar que qualquer aplicação possa empregar o serviço ao estruturar uma replicação.

3.3. Modularidade

A arquitetura prevê que o consenso seja um módulo acoplável, no qual é possível optar por um serviço de *logs* replicados ou por uma implementação com protocolos de consenso para assegurar a ordenação das requisições. Este módulo visa, além da ordenação, reter as requisições por um tempo configurável até o envio para as réplicas. Além da modularidade prevista na implementação do consenso, a própria proposta é solução modular para a implementação da replicação nas aplicações.

3.4. Escalabilidade

Como o serviço propõe o desacoplamento de toda estratégia de replicação das réplicas, é possível escalar aplicação com facilidade, visto que para adição de servidores não é necessário que a aplicação seja configurada para adentrar no ecossistema. Além disso, pretende-se investigar a possibilidade de aplicar estratégias de *sharding* e particionamento, de modo que requisições de clientes possam ser destinadas para conjuntos de réplicas independentes. O particionamento, entretanto, pode restringir o grau de transparência, pois requer conhecimento específico sobre a semântica do serviço a ser replicado.

4. Implementação

O serviço de replicação está em estágio de revisão e discussões da literatura, entretanto, há intenção do desenvolvimento de um protótipo de um mediador responsável por interceptar as requisições de um cliente e servidor genéricos e que envie estas para uma instância do Apache BookKeeper, serviço que provê *logs* distribuídos com replicação e foi, inicialmente, eleito para o desenvolvido do protótipo.

Após esse desenvolvimento, pretende-se aplicar ao protótipo outras estratégias para assegurar a ordenação (como uma implementação do Paxos), com objetivo de comparar o desempenho entre elas e certificar a modularidade do serviço. Por fim, o serviço proposto irá prosseguir para a etapa de avaliação, para guiar as modificações necessárias até o desenvolvimento do projeto definitivo.

5. Conclusão

Devido à constante evolução e transmissão contínua de informações na sociedade contemporânea, a garantia da disponibilidade de serviços é crucial. Logo, a replicação em múltiplos servidores é uma estratégia para mitigar os prejuízos decorrentes da indisponibilidade, proporcionando confiabilidade e redundância. Entretanto, há desafios associados à replicação passiva e à Replicação de Máquina de Estados. Portanto, a proposta foca no desenvolvimento de um serviço de replicação desacoplado que certifique a ordenação das requisições, atue na interceptação das requisições e como gerenciador das respostas para os clientes. Com isso, o projeto visa superar algumas limitações de outras soluções ao proporcionar desacoplamento, generalização, flexibilidade modular e escalabilidade.

Referências

Budhiraja, N., Marzullo, K., Schneider, F. B., and Toueg, S. (1993). The primary-backup approach. In *Distributed Systems (2nd Ed.)*. ACM Press/Addison-Wesley Publishing Co.

Coelho, F., Alonso, A., Ferreira, L., Pereira, J., and Oliveira, R. (2023). Loom: A closed-box disaggregated database system. In *Proceedings of the 12th Latin-American Symposium on Dependable and Secure Computing*, pages 30–39.

Guo, S., Dhamankar, R., and Stewart, L. (2017). Distributedlog: A high performance replicated log service. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 1183–1194.

Independent, T. (2023). Amazon down: Internet outage hits sales. https://www.independent.co.uk/news/business/amazon-down-internet-outage-sales-b1861737.html.

Scharf, J. a. L., Xavier, L. G. C., and Mendizabal, O. M. (2023). Joining parallel and partitioned state machine replication models for enhanced shared logging performance. In *Proceedings of the 12th Latin-American Symposium on Dependable and Secure Computing*, page 90–99.

Schneider, F. B. (1990). Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319.

Verbitski, A., Gupta, A., Saha, D., Brahmadesam, M., Gupta, K., Mittal, R., Krishnamurthy, S., Maurice, S., Kharatishvili, T., and Bao, X. (2017). Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *SIGMOD 2017*.

Xavier, L. G., Dotti, F., Meinhardt, C., and Mendizabal, O. (2020). Scalable and decoupled logging for state machine replication. In *Anais do XXXVIII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, pages 267–280.