UNIVERSIDADE FEDERAL DE SANTA CATARINA
CAMPUS FLORIANÓPOLIS
BACHARELADO EM SISTEMAS DE INFORMAÇÃO

Lucas Pagotto Tonussi

Estendendo um Ordenador de Mensagens em Arquitetura de Microsserviços para Comunicação sobre o Protocolo HTTP

Florianópolis (SC)
2022

Lucas Pagotto Tonussi

Estendendo um Ordenador de Mensagens em Arquitetura de Microsserviços para Comunicação sobre o Protocolo HTTP

Monografia submetida ao Bacharelado em Sistemas de Informação da Universidade Federal de Santa Catarina para a obtenção do título de Bacharel em Sistemas de Informação.
Orientador: Prof. Odorico Machado Mendizabal, Dr.

Florianópolis (SC)
2022

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

Tonussi, Lucas Pagotto
Estendendo um Ordenador de Mensagens em Arquitetura de Microsserviços para Comunicação sobre o Protocolo HTTP / Lucas Pagotto Tonussi ; orientador, Odorico Machado Mendizabal, 2022.
49 p.

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

Inclui referências.

1. Sistema de Informação. 2. Arquitetura de microsserviços. 3. Ordenação de mensagens. 4. Kubernetes. 5. Interceptação de mensagens. I. Machado Mendizabal, Odorico. II. Universidade Federal de Santa Catarina. Graduação em Sistema de Informação. III. Título.

Lucas Pagotto Tonussi

Estendendo um Ordenador de Mensagens em Arquitetura de Microsserviços para Comunicação sobre o Protocolo HTTP

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

Prof.(a) Luciana de Oliveira Rech, Dr(a).
Universidade Federal de Santa Catarina

Prof.(a) Alex Sandro Roschildt Pinto, Dr(a).
Universidade Federal de Santa Catarina

Certificamos que esta é a **versão original e final** do trabalho de conclusão que foi julgado adequado para obtenção do título de Bacharel em Sistemas de Informação.

Coordenação do Programa de
Pós-Graduação

Prof. Odorico Machado Mendizabal, Dr.
Orientador

Florianópolis (SC), 2022.

Este trabalho é dedicado aos meus amigos íntimos,
irmãos, colegas de classe e aos meus queridos pais.

AGRADECIMENTOS

Agradeço ao meu orientador Odorico Machado Mendizabal pela mentoria, ensinamentos, orientações e revisões do texto. Agradeço também ao Renan Tarouco da Fonseca pelas muitas ajudas e a Marina Milhomens Queiroz pela ajuda com a escrita.

“Thinking doesn’t guarantee that we won’t make mistakes. But not thinking guarantees that we will”
(LAMPORT L., 2013)

RESUMO

Recentemente, arquiteturas baseadas em microsserviços ganharam popularidade, em parte por causa do modelo de programação modular, acoplamento mínimo entre as partes e o suporte de plataformas de orquestração de contêineres. Ordenação de mensagens é uma estratégia que garante que todas as réplicas evoluam igualmente, aumentando-se os níveis de disponibilidade de serviços. Visando aplicações que usufruem de interfaces HTTP para operar, este trabalho propõe uma implementação de interface de comunicação, sobre o protocolo HTTP e para um ordenador de mensagens. Sabe-se que orquestradores de contêineres oferecem replicação de forma automática, porém o serviço oferecido por orquestradores garante replicação de aplicações *stateless*. O objetivo é continuar desenvolvendo um ordenador de mensagens transparente ao usuário, para isto este trabalho estende uma pesquisa iniciada pelo grupo, que propõe o Hermes, um interceptador de mensagens como serviço que usufrui de mecanismos de orquestração de contêineres para prover replicação e tolerância a falhas. O desenvolvimento do serviço de ordenação de mensagens contou com a implementação da interface que promove a comunicação no Hermes. A implementação possibilita que o Hermes possa tratar mensagens HTTP. Ao final houve investigação de desempenho da implementação em casos específicos de vazão e latência. Os experimentos incluíram duas aplicações para avaliação de desempenho: uma aplicação de *log* que recebe requisições HTTP e salva, em arquivo de disco e uma aplicação geradora de carga que envia requisições HTTP, podendo ser configurada por parâmetros. A investigação demonstrou que as latências capturadas nos geradores de carga apresentaram valores maiores para o sistema replicado quando comparado com o caso não-replicado, isto era esperado. O cenário de carga de 100% POST, os experimentos se mostraram mais promissores. O caso onde existe 100% de cargas GET os experimentos se mostraram melhores que no caso híbrido de 50% GET e 50% POST, por causa que existe 50% de chance de múltiplos processos inserirem mais linhas no arquivo de *log*. Finalmente, as comparações entre os cenários replicados e não-replicado mostraram que o ordenador de mensagens prove tolerância a falhas e replicação ativa de aplicações *stateful* baseadas em HTTP.

Palavras-chave: Protocolo de consenso. Proxy. Interceptação de Mensagens. Orquestração de contêineres. Ordenação total. Arquitetura de microsserviços. Kubernetes. Docker.

ABSTRACT

Recently, architectures based on micro-services gained popularity, in part because of the modular programming model, minimum coupling between parts of the system and support on container orchestration platforms, also offering automatic resources management. Message ordering is a strategy that guarantees that all replicas of a distributed system will evolve equally, raising the levels of availability of services and fault tolerance. Aiming applications that use HTTP to operate, this work proposes a implementation of a interface of communication over the HTTP protocol and for a message ordering system. It's known that container orchestrators offers replication automatically, although that replication works, adequately, for stateless applications. This proposal aims to cover cases where the server being replicated is stateful. The objective is to continue the development of a transparent ordering system, for that the present work extends a work previously proposed by the research group, a work called Hermes, which is a service that intercepts messages and take advantage of a container orchestration system. The improvement of that message orderer service counted on a new implementation of the interface that covers communication, the implementation enables the message orderer system to handle HTTP requests. Afterwards, benchmarks of throughput and latency were made. The experiments included two applications: a log application that receives HTTP and saves, in a disk file, the request body and a stress generator application that sends HTTP requests, and can be configured by parameters. The experiments showed that the latencies captured on the stress generators received greater numbers when compared to the non-replicated case, that was expected. The scenario where there is a 100% load of POST requests, the experiment showed a more promising scenario. The case where there is a 100% load of GET requests, the experiment showed better results than the 50% GET 50% POST mixture, because there is a 50% chance of multiple processes inserting more lines into the file. Finally the comparisons between the replicated and non-replicated scenarios showed that the Hermes is able to tolerate failures and replicate HTTP based stateful applications.

Keywords: Consensus protocol. Proxy. Message intercepting. Container orchestration. Total ordering. Micro-services architectures. Kubernetes. Docker.

LISTA DE FIGURAS

LISTA DE ALGORITMOS

LISTA DE ABREVIATURAS E SIGLAS

AB-cast	*Atomic Broadcast*
ACM	*Association for Computing Machinery*
API	*Application Programming Interface*
ASCII	*American Standard Code for Information Interchange*
FIFO	*First In First Out*
HTTP	*Hypertext Transfer Protocol*
IEEE	*Institute of Electrical and Electronics Engineers*
IP	*Internet Protocol*
ROX	*Read Only Many*
RWO	*Read Write Once*
RWX	*Read Write Many*
SBC	Sociedade Brasileira de Computação
UDP	*User Datagram Protocol*

SUMÁRIO

1 INTRODUÇÃO

As arquiteturas de microsserviços têm recebido grande atenção para o desenvolvimento de aplicações distribuídas e vêm sendo amplamente adotadas, especialmente em provedores de computação em nuvem (AGUILERA *et al.*, 2020; NETTO *et al.*, 2020; TAI, 2016; MOGHADDAM; KANSO; GHERBI, 2016; NGUYEN, N.; KIM, 2020; GABBRIELLI *et al.*, 2016; TOFFETTI *et al.*, 2015; OLIVEIRA *et al.*, 2016). Em particular, o desenvolvimento de serviços nessas arquiteturas com suporte de orquestradores de contêineres facilita o gerenciamento de escalabilidade, reaproveitamento de recursos e integração contínua (GHOFRANI; LÜBKE, 2018).

Observa-se uma tendência em implementar sistemas de replicação mais eficientes com microsserviços, visando aliar a praticidade de desenvolvimento baseado em arquiteturas de microsserviços e consistência forte dos dados (TOFFETTI *et al.*, 2015). Apesar de orquestradores, como o Kubernetes, proverem mecanismos para replicação de contêineres com muita facilidade, não há garantia de que o estado de uma réplica seja mantido idêntico ao estado das demais réplicas, dado que as réplicas modificam seus estados internos independentemente. Portanto, a garantia de consistência fica por conta do desenvolvedor de software.

Em um trabalho recente, Fonseca (2021) apresentou uma prova de conceito relacionando tolerância a falhas e a adoção de arquitetura de microsserviços, chamada Hermes. Trata-se de um ordenador de mensagens à frente de cada réplica de um sistema distribuído, com orquestração de contêineres. O serviço proposto utiliza protocolos de consenso para estabelecer um acordo uniforme entre as réplicas com respeito a próxima requisição que deve ser executada. Este serviço serve como um bloco de construção para prover ordenação de mensagens, onde cada réplica parte de um mesmo estado inicial, recebem as mesmas requisições de clientes e processam as requisições de forma determinística e em uma mesma ordem, garantindo que as réplicas estejam consistentes. Atualmente a utilização do serviço de ordenação proposto para a implementação de ordenação de mensagens permite apenas ordenação de mensagens sobre *sockets* TCP.

Dessa forma, o presente trabalho estende o trabalho de Fonseca (2021). A ideia é manter um conjunto de microsserviços frente às réplicas da aplicação e apresentar um novo tipo de protocolo de comunicação, no caso HTTP, para o Hermes. Esta abordagem pode ampliar a adoção de serviço de ordenação, contemplando aplicações de terceiros que utilizam comunicação via HTTP.

1.1 MOTIVAÇÃO

O presente trabalho visa explorar a extensibilidade do Hermes e fazer experimentações em um *cluster* real, estudando a implantação do Hermes. O trabalho de

Fonseca (2021) foi implementado em linguagem Go e o Hermes usa comunicação TCP. Contudo, o presente trabalho se motivou em adicionar a possibilidade do ordenador de mensagens se comunicar via HTTP, ampliando a possibilidade de aplicações se ligarem ao Hermes.

1.2 OBJETIVO

Apresentar uma implementação alternativa de comunicação em protocolo HTTP para o interceptador de mensagens Hermes (FONSECA, 2021). Dentre os objetivos específicos, pode-se citar:

- *Meio de comunicação:* Implementar a interface de comunicação do Hermes para possibilitar requisições HTTP.
- *Implementar aplicações de exemplo para testar a integração destas com o serviço de replicação:* Para este propósito, foram desenvolvidas aplicações como *log* em disco.
- *Avaliar a escalabilidade da implementação em protocolo HTTP:* Para isto foram feitas experimentações em ambiente real, extraindo métricas de análise de desempenho.

1.3 ORGANIZAÇÃO DO TRABALHO

O presente trabalho está organizado da seguinte forma: o Capítulo 2 apresenta a fundamentação teórica; o Capítulo 3 apresenta os trabalhos correlatos; o Capítulo 4 apresenta o interceptador de mensagens Hermes; o Capítulo 5 apresenta a implementação do protocolo HTTP no Hermes; o Capítulo 6 apresenta a avaliação experimental; o Capítulo 7 apresenta a conclusão do trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

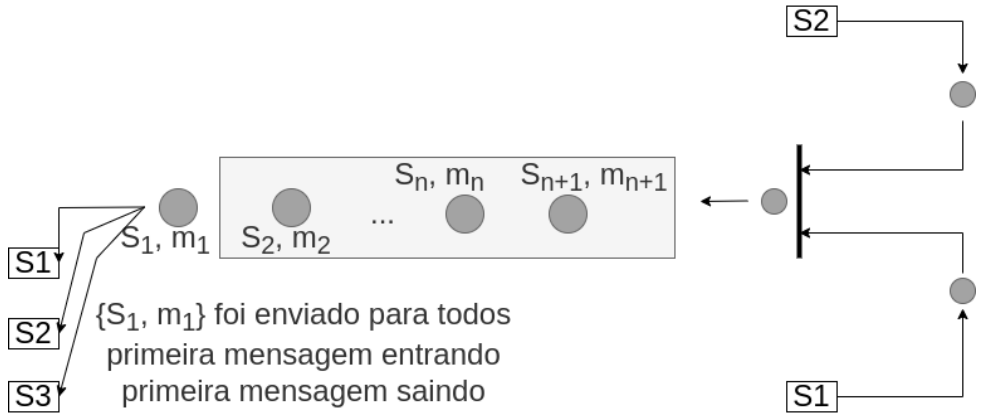
2.1 ORDENAÇÃO DE MENSAGENS

Um ordenador de mensagens é um mecanismo vital para promover coerência, em ordenação de eventos entre os participantes de um sistema distribuído e em um sistema replicado, usufruindo-se de replicação ativa, a ordenação de mensagens se torna vital para que o sistema evolua de maneira igual. Existem na literatura várias técnicas de ordenação de mensagens. A seguir são apresentados alguns tipos de ordenação de mensagens.

2.1.1 Ordem FIFO

A ordenação por *First-In First-Out* (FIFO), que significa o primeiro a entrar é o primeiro a sair, estabelece que as mensagens enviadas pelo mesmo participante e entregues para qualquer participante, são entregues na mesma ordem de envio (VERÍSSIMO; RODRIGUES, 2001).

Figura 1 – Servidores S1 e S2 enviando mensagens FIFO



Fonte - Própria

A Figura 1 representa uma ideia de como o esquema de ordenação de mensagens FIFO pode funcionar. A ilustração em si demonstra que existe uma fila, que deve ser implementada por um ordenador de mensagens, assim as mensagens podem ser enfileiradas, ao passo que são enfileiradas, as mensagens também são consumidas na saída, dando vazão para as mensagens.

2.1.2 Ordem Causal

A ordem causal, definida por Lamport (LAMPORT, 1978), estabelece algumas premissas: O símbolo *m* é uma mensagem qualquer que acompanha um ID único e $m_1, m_2, \ldots, m_n \in M$, onde M é um universo de mensagens; *send*(*m*) é uma primitiva

que executa o envio da mensagem $m \in M$; $p, q, r \in$ *participantes* onde os participantes são as entidades do sistema, que podem trocar mensagens entre si; *deliver*(m) é uma primitiva de entrega de mensagens.

Na ordem causal, existe a definição da propriedade *acontece antes*, simbolizado por $\longrightarrow$ e é um operador que designa uma ordem de precedência entre as mensagens.

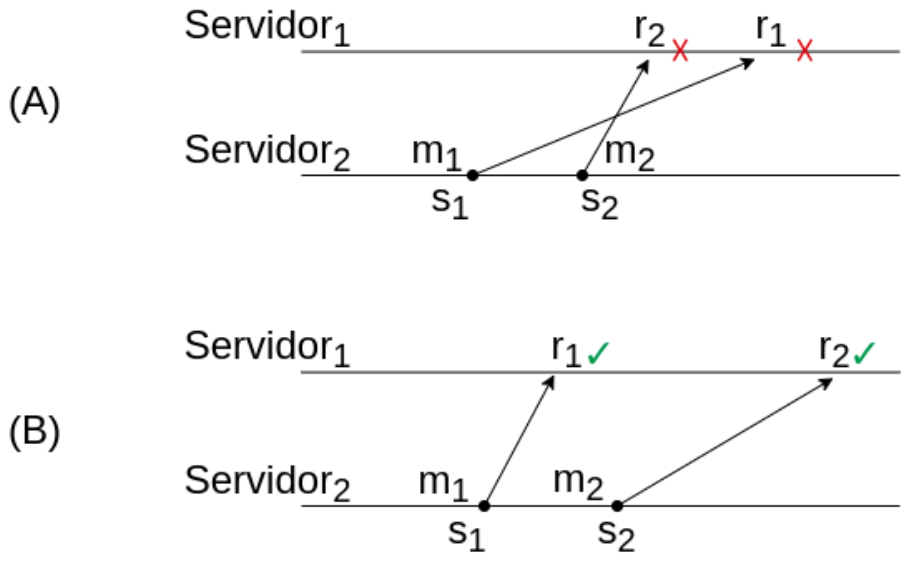
Por ordem lógica, define-se: Uma mensagem m_1 precede logicamente m_2 *i.e.* $m_1 \longrightarrow m_2$, sse: m_1 é enviada antes de m_2 pelo mesmo participante **ou** m_1 é entregue para o *enviador* de m_2 antes que ele envie m_2 **ou** existe um m_3 tal que $m_1 \longrightarrow m_3$ e $m_3 \longrightarrow m_2$ (VERÍSSIMO; RODRIGUES, 2001).

Exemplificando: Se um evento *a* aconteceu antes do evento *b* *i.e.* $a \longrightarrow b$, então o evento *b* pode ter sido causado ou influenciado pelo evento *a*. Se *a* e *b* são duas mensagens *multicast* (respeitando a causalidade), tem-se que a entrega de *a* deve preceder a entrega de *b*, em todos os destinos comuns de *a* e *b*. Abaixo está a especificação de entrega de pedido causal em grupos sobrepostos (ARAÚJO MACÊDO, 1995).

> *Definição de Ordem Causal*:
> $send_p(m_1) \longrightarrow send_q(m_2)$ então
> $deliver_r(m_1) \longrightarrow deliver_r(m_2)$, *i.e.* m_1
> é entregue para r antes de m_2

A ordem causal pode ser implementada usando relógios lógicos de Lamport (LAMPORT, 1978) para capturar o sentido de causalidade sobre as entregas de mensagens.

Figura 2 – Exemplos de ordem causal



Fonte - Adaptado de Veríssimo e Rodrigues (2001)

A Figura 2 mostra dois exemplos (A e B) de ordem causal, os símbolos podem ser lidos da seguinte forma: m_1 e m_2 são mensagens, s_1 e s_2 são envios das mensagens, r_1 e r_2 são recebimentos das mensagens, $Servidor_1$ e $Servidor_2$ são os servidores do sistema distribuído. O exemplo (A) viola a ordem causal pois a mensagem m_2 chegou antes de m_1, sendo que $Servidor_2$ enviou primeiro m_1 e depois enviou m_2, ou seja, m_1 deve acontecer antes de m_2, pois existe uma relação de causa e ordem que deve ser obedecida. Essa relação pode ser observada no exemplo (B).

2.1.3 Ordem Total

Na ordenação total todos os participantes recebem as mensagens enviadas na mesma ordem. Porém, em termos mais técnicos, pode-se definir a partir de Veríssimo e Rodrigues (2001) que:

> *Quaisquer duas mensagens entregues para quaisquer dois pares de participantes são entregues na mesma ordem para ambos os participantes.*

O termo difusão em ordem total é também conhecido como difusão atômica, algumas vezes representado pela sigla AB-cast. A seguir são apresentada as propriedades necessárias para assegurar ordenação total e os algoritmos para implementação de protocolo AB-cast.

2.2 PROTOCOLOS PARA IMPLEMENTAÇÃO DE ORDEM TOTAL

Define-se o problema com basicamente 2 primitivas e o conceito de processos corretos. As primitivas são *TO-broadcast(m)* e *TO-deliver(m)*, onde $m \in M$ é alguma mensagem que pode ser unicamente identificada e, que carrega a identificação do remetente, denotada por *sender(m)* (VERÍSSIMO; RODRIGUES, 2001).

A primitiva *TO-broadcast* expressa a execução de difusão de uma mensagem para todos os processos corretos de um sistema, ou seja, todos esses processos corretos recebem as mesmas mensagens e na mesma ordem. *TO-deliver* é outra primitiva que é executada sempre após *TO-broadcast* (VERÍSSIMO; RODRIGUES, 2001).

Os processos corretos pertencem a uma categoria de processos que não falham no sistema. Por exemplo, são processos que não omitem ou param de performar suas atividades de envio e recebimento de mensagens.

A especificação básica para ordenação total, conforme descrito por Hadzilacos e Toueg (1994), é apresentada a seguir:

- **Validade**: Se um processo correto executa *TO-broadcast* sobre uma mensagem *m*, então vai eventualmente executar *TO-deliver* para a mensagem *m*.

- **Acordo Uniforme**: Se um processo executa *TO-deliver* sobre uma mensagem *m*, então todos os processos corretos irão eventualmente executar *TO-deliver* sobre *m*.
- **Integridade Uniforme**: Para qualquer mensagem *m*, cada processo executa *TO-deliver* sobre uma mensagem *m*, pelo menos 1 vez, e apenas se previamente fora executado *TO-broadcast* sobre uma mensagem *m* por *sender(m)*.
- **Ordem Total Uniforme**: Se ambos os processos *p* e *q* executarem *TO-deliver* sobre a mensagens *m* e $m^{'}$, então *p* executa *TO-deliver* sobre *m* antes de $m^{'}$, sse *q* executa *TO-deliver* sobre *m* antes de $m^{'}$.

2.2.1 Ordenação por sequenciador

O algoritmo de ordenação por sequenciador (DÉFAGO; SCHIPER, André; URBÁN, 2004) tem a incumbência de ordenar todas as mensagens, sendo que para cada mensagem é associado um número de sequência. O algoritmo tem 3 papéis: o remetente (Algoritmo 1), o sequenciador (Algoritmo 2) e os destinatários (Algoritmo 3).

Algoritmo 1: Código dos remetentes para o algoritmo de sequenciador fixo simples

```
1 procedure TO-broadcast(m)
2     send(m) to sequencer
```

O Algoritmo 1 apresenta a chamada do procedimento TO-broadcast(m). A primitiva TO-broadcast(m) é a representação da difusão por ordem total. Dentro do escopo do procedimento se executa a primitiva *send(m)* ao sequenciador, discutido a seguir.

Algoritmo 2: Código do sequenciador para o algoritmo de sequenciador fixo simples

```
1 seqnum := 1
2 when receive(m)
3     sn(m) := seqnum
4     send(m, sn(m)) to all
5     seqnum := seqnum + 1
```

O Algoritmo 2 representa o tratamento constante que é dado para mensagens *m* que estão sendo recebidas. Começa com o número de sequência que recebe valor 1, esse valor é configurado uma vez apenas. Depois disto, o algoritmo só trata quando a mensagem *m* existe e é recebida pela primitiva *receive*. O escopo de *when receive(m)* tem 3 atribuições: na linha 3 o número de sequência de *m* recebe o valor de *seqnum* atual; na linha acontece o envio da mensagem *m* junto com seu número de sequência

através da primitiva *send(m, sn(m))* para todos os participantes; na linha 5 ocorre o incremente do número de sequência.

Algoritmo 3: Código dos destinatários para o algoritmo de sequenciador fixo simples

Input: Código do processo p_i

1 $nextdeliver_{p_i} := 1$
2 $pending_{p_i} := \emptyset$
3 **when** *receive (m, seqnum)*
4 $\quad pending_{p_i} := pending_{p_i} \cup \{(m, seqnum)\}$
5 $\quad$ **while** $\exists(m', seqnum') \in pending_{p_i}$ *s.t.* $seqnum' = nextdeliver_{p_i}$
6 $\quad\quad$ deliver(m')
7 $\quad\quad nextdeliver_{p_i} := nextdeliver_{p_i} + 1$

O Algoritmo 3 inicializa os conjuntos de dados: *nextdeliver* com todos os valores igual à 1, e *ending* como vazio. Por conseguinte o algoritmo começa a processar continuamente o recebimento de mensagens, sendo que cada mensagem vem acompanhada de seu *seqnum* (número de sequência). O primeiro passo dentro do escopo do *when*, linha 4, implementa a união de novas mensagens no conjunto *pending*. Quando o fluxo atinge a linha 5, ocorre um laço repetitivo. Neste laço repetitivo se verifica a existência de uma mensagem *m* com *seqnum* pertencente ao conjunto *pending*, se for o caso então *seqnum* receberá o valor de *nextdeliver[p_i]* (final da linha 5). Após isso duas linhas de código são executadas, na linha 6 ocorre a execução da primitiva *deliver(m)*. E na linha 7, ocorre o incremento de *nextdeliver* na posição de p_i.

2.2.2 Ordenação baseada em privilégio

O algoritmo baseado em privilégio (DÉFAGO; SCHIPER, André; URBÁN, 2004) tem 2 papéis principais: os remetentes (Algoritmo 4) e os destinatários (Algoritmo 5). A intenção geral do algoritmo é implementar a difusão atômica, para isso, rotaciona-se um *token* entre os remetentes. Quando cada remetente recebe o *token*, na sua vez, executa-se a primitiva *tosend*, enviando para todos os destinatários as mesmas mensagens e na mesma ordem de chegada.

Algoritmo 4: Código dos remetentes para o algoritmo baseado em privilégios

```
   Input: Código do processo s_i
1  tosend_si := ∅
2  if s_i = s_1 then
3      token.seqnum := 1
4      send token to s_1
5  procedure TO-broadcast(m)
6      tosend_si ∪ {m}
7  when receive token
8      for each m′ in tosend_si
9          send(m′, token.seqnum) to destinations
10         token.seqnum := token.seqnum + 1
11     tosend_si := ∅
12     send token to s_{i+1 mod n}
```

O Algoritmo 4 inicializa o conjunto *tosend* como vazio. Se o processo s_i é o primeiro, então inicializa *token.seqnum* com 1 (linha 3) e, executa-se a primitiva *send(token)* para s_1. Na linha 5 executa-se o procedimento de difusão ordem total sobre a mensagem *m*, na linha 6 a mensagem *m* é unida ao conjunto *tosend*. Na linha 7 quando o código dos remetentes recebe um novo token, executa-se, na linha 8, um laço iterativo, ou seja, para cada *m* dentro do conjunto *tosend*, executa-se, na linha 9, a primitiva *send* para a mensagem *m* junto com o valor atual de token.seqnum (rotacionando) para os destinatários. Na linha 10 a variável *token.seqnum* recebe um incremento de 1. Fora do laço repetitivo, na linha 11, o conjunto *tosend* recebe vazio. Na linha 12 executa-se a primitiva *send* sobre o objeto *token* para o processo *s* porém é feito um cálculo modular para iterar pelos processos participantes de forma anelar. Uma vez que o próximo processo obtêm o *token*, é a vez do processo recebedor de executar o algoritmo.

Algoritmo 5: Código dos destinatários para o algoritmo baseado em privilégios

```
   Input: Código do processo p_i
1  nextdeliver_pi := 1
2  pending_pi := ∅
3  when receive (m, seqnum)
4      pending_pi := pending_pi ∪ {(m, seqnum)}
5      while ∃(m′, seqnum′) ∈ pending_pi s.t. seqnum′ = nextdeliver_pi
6          deliver m′
7          nextdeliver_pi := nextdeliver_pi + 1
```

O Algoritmo 5 e o Algoritmo 3 são iguais e portanto têm as mesmas explicações.

2.3 CONSENSO

Protocolos de consenso distribuído também podem ser utilizados para implementar entrega totalmente ordenada de requisições (EKWALL; SCHIPER, André, 2006; MILOSEVIC; HUTLE; SCHIPER, Andre, 2011). Esses protocolos surgem como uma boa alternativa, visto que a difusão atômica pode gerar uma quantidade muito grande de requisições. Segundo Lamport *et al.* (2001), os algoritmos de consenso precisam garantir 3 premissas básicas:

- **Não trivialidade**: Um valor é aprendido se e somente se tiver sido proposto e aceito pela maioria dos participantes.
- **Estabilidade**: A cada rodada de aprendizagem, qualquer processo participante deve aprender no máximo um valor. Um valor aprendido não deve ser alterado posteriormente pois um valor aprendido será executado na aplicação.
- **Consistência**: A cada rodada de aprendizagem, dois processos diferentes não devem aprender valores diferentes.

2.3.1 Protocolos de Consenso

Protocolos de consenso são centrais para a implementação de aplicações distribuídas tolerantes a falhas (CASON, 2017). O problema de consenso surgiu da necessidade de múltiplos processos chegarem a um acordo sobre um determinado valor. Portanto, participantes de um sistema distribuído necessitam de um algoritmo capaz de estabelecer um quórum decisivo sobre um valor proposto pelo algoritmo de consenso.

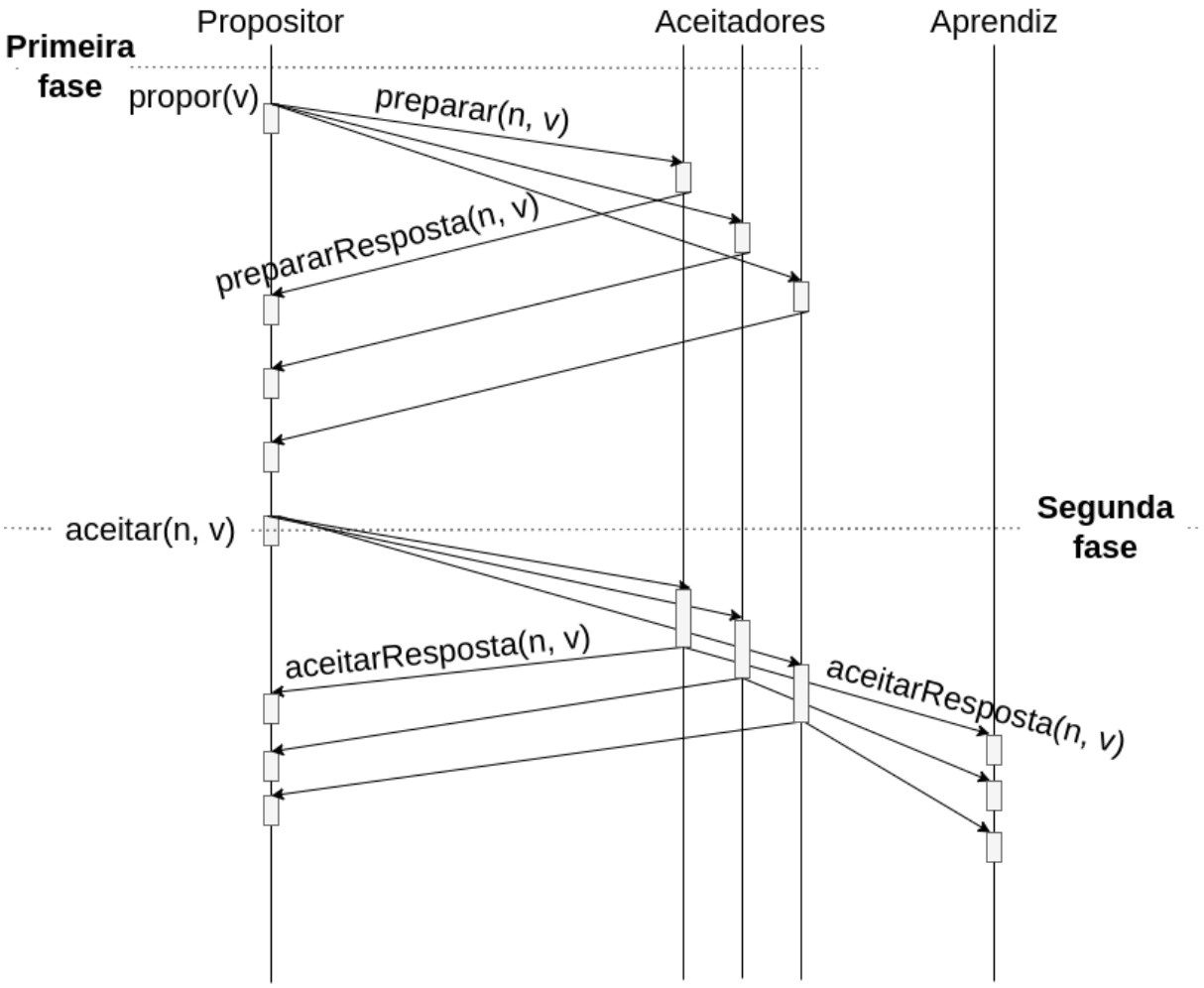
2.3.1.1 Paxos

O algoritmo Paxos foi desenvolvido por Lamport *et al.* (2001) e geralmente se implementa Paxos entre um conjunto distribuído de computadores trocando mensagens de forma assíncrona. Por exemplo, um cliente envia um valor ao sistema e o protocolo Paxos é invocado para propor este valor aos participantes. Quando a maioria dos participantes que executam o Paxos concordarem sobre o valor proposto, então este valor será aprendido e entregue aos interessados neste valor, por exemplo, a um conjunto de réplicas. Existem 3 papéis no algoritmo Paxos:

- *Proponente:* Recebe solicitações de valores externos e as propõem aos aceitadores. Cada proposta nova deve conter um número único identificador e maior que a proposta anterior.
- *Aceitadores:* Aceitam determinado valor proposto pelo proponente e informam ao proponente e aos aprendizes se algum valor foi aceito. Uma resposta de um aceitador representa um voto para uma proposta.

- *Aprendizes:* Aprendem o resultado do consenso, se houver.

Figura 3 – Exemplo de troca de mensagens no protocolo Paxos



Fonte - Adaptado de Terra, Camargo e Jr. (2020)

O Paxos tem basicamente duas fases e a Figura 3 ilustra as fases 1ª e 2ª. As fases foram subdivididas em a-b e que serão brevemente discutidas, a seguir.

1. *Primeira fase: Proposta e Preparação*

    a) O protocolo se inicia quando um proponente recebe um valor *v*. O valor *v* pode ser, por exemplo, um comando que será executado em um determinado momento (no caso a natureza do valor *v* depende muito da aplicação *stateful*). O proponente propõe um valor *v* enviando uma mensagem igual para cada um dos aceitadores. O algoritmo envia uma mensagem *v* por proposta e está proposta terá outra difusão de mensagens para cada um dos aceitadores. Cada mensagem contém o valor da proposta e um número de sequência. O número de sequência é um identificador da proposta e quando o

propositor propõe um valor *v*, este também prepara as mensagens e envia aos aceitadores.

b) Os aceitadores avaliam a proposta de valor e emitem suas respostas ao propositor, sinalizando que reconheceram a mensagem *m* (que acompanha um número de sequência como identificador, juntamente com o valor proposto). Os aceitadores só devem aceitar a proposta com o número de sequência maior do que qualquer outro. A proposta já aceita com número identificador menor que o atual serão descartadas.

2. *Segunda fase: Aceitação e Aprendizado*

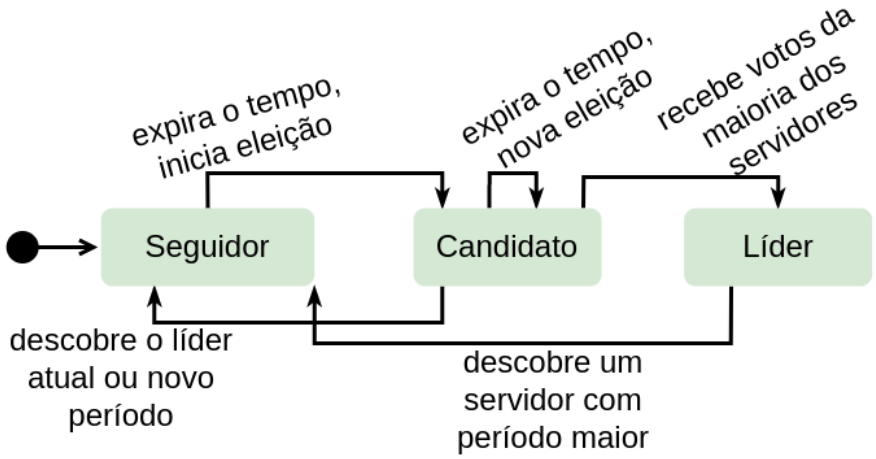
a) Os aceitadores recebem uma segunda rodada de mensagens, onde o propositor envia as mensagens, nesse estágio o propositor envia uma sequência de mensagens para aceitar o valor *v* com o número de sequência *n*, ou seja, os aceitadores recebem *aceitar(n, v)*.

b) O aprendiz aprende se os aceitadores chegaram a um consenso em uma rodada específica do algoritmo. A quantidade de aceitadores precisa ser pelo menos metade mais um, do conjunto de servidores do sistema. Se a maioria dos aceitadores responderem que concordam com o valor *v*, então *v* será aprendido.

2.3.1.2 Raft

O Raft é um algoritmo distribuído e assíncrono de ordenação de eventos (ONGARO; OUSTERHOUT, 2014). Este algorítimo espera que exista um sistema de replicação de *logs* em cada instância que executa o protocolo, e cada instância do sistema Raft pode estar em um dos seguintes estados: Líder, Candidato, Seguidor, porém só existe um líder por vez e o líder recebe requisições do cliente e pode propor mensagens aos seguidores.

Figura 4 – Fases de um serviço Raft



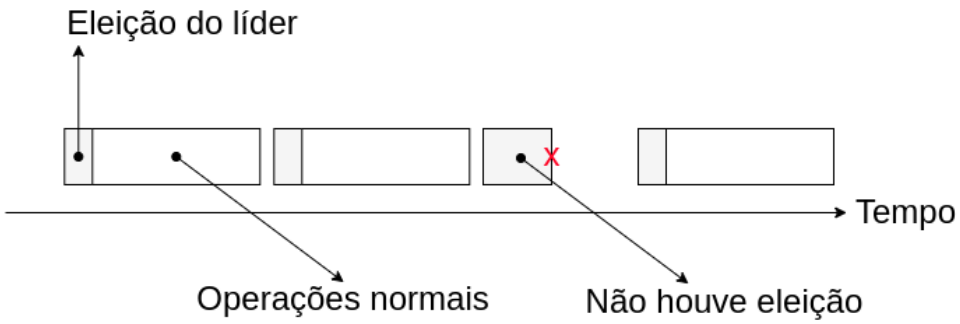
Fonte - Adaptado de (ONGARO; OUSTERHOUT, 2014)

A Figura 4 ilustra os estados de cada nó participante e as possíveis transições. Os seguidores reagem às propostas do líder. Se os candidatos não recebem mais sinais do líder, elege-se um novo líder dentre os candidatos. Se algum candidato estiver no estado de seguidor, este deve poder votar por propostas que chegam ou votar para a eleição de novos líderes. É responsabilidade do líder fazer a replicação dos *logs*. Deve haver apenas 1 líder por termo. O líder é responsável pela replicação de *logs*.

Ongaro e Ousterhout (2014) buscaram desenvolver o algoritmo Raft com técnicas que aumentassem a legibilidade do algoritmo. Algumas das técnicas utilizadas foram: decomposição do problema em pequenas partes, minimização do espaço de estados, tratar múltiplos problemas com um único mecanismo, eliminação de casos especiais e minimização de não-determinismo.

A Figura 5 ilustra um exemplo do funcionamento dos termos em Raft. Os *Termos* em Raft existem para que o Raft possa identificar informação obsoleta e distinguir de informação nova. Cada máquina do *cluster* mantêm o seu *termo* atual e não são compartilhados globalmente. Cada *Termo* se trata de um período de tempo onde ocorre uma eleição de um líder e após isso ocorrem proposta. Um *Termo* é subdividido em 2 fases:

Figura 5 – Termos em Raft



Fonte - Adaptado de Ongaro e Ousterhout (2014)

Primeira fase: será feita a escolha de um líder.
Segunda fase: serão feitas propostas de mensagens aos seguidores.

2.4 ARQUITETURA DE MICROSSERVIÇOS

Publicações recentes mostram que implementar em arquiteturas de microsserviços provê desacoplamento e escalabilidade (CARRASCO; BLADEL; DEMEYER, 2018; BAŠKARADA; NGUYEN, V.; KORONIOS, 2018; ZDUN; WITTERN; LEITNER, 2019). O trabalho de Fritzsch *et al.* (2018) provê dez abordagens estratégicas de refatoração, viabilizando migrar monólitos para microsserviços. A implementação de software seguindo arquitetura de microsserviços requer desenvolvimento de partes modulares, onde cada módulo pode ser conectado e desconectado em outros sistemas, sem que haja necessidade de mudar o sistema principal que faz uso do microsserviço. Geralmente, os microsserviços são enxutos com *interfaces* específicas que viabilizam extensibilidade de funcionalidades específicas (JAMSHIDI *et al.*, 2018).

2.4.1 Contêineres e Orquestradores de contêineres

Um contêiner se trata de um ambiente empacotado de: software, configurações, dependências além de outras coisas para tornar possível que um serviço seja executado isoladamente de outros softwares da máquina hospedeira.

Um orquestrador é um gestor de contêineres com características de auto-cura e elasticidade. Também pode ser usado para a criação de *cluster* (conjunto de máquinas interligadas por uma rede). Dependendo do fabricante da tecnologia algumas características podem mudar, porém geralmente o orquestrador tem características de: escalonamento de réplicas baseado em métricas, balanceamento de carga. Orquestradores podem operar sob os contêineres em grupos específicos (CASALICCHIO, 2019), descomplicando de se aplicar atribuições específicas para cada grupo de contêineres e podendo-se criar divisões lógicas que aprimoram a especificação do *cluster* como um todo.

2.4.2 Docker

Um contêiner Docker[1] (DOCKER, 2022) é uma instância executável de uma imagem. Uma imagem do Docker é um modelo *read only* que provê instruções para construção de um contêiner Docker. O Docker é uma tecnologia de código-aberto que oferece: preparo, entrega e execução de aplicativos auto-contidos em contêineres. Com Docker é possível: criar, iniciar, parar, mover e excluir um contêiner usando a API do Docker. O Docker integra com funcionalidades Linux como *cgroups* e *namespaces* para usufruir dos recursos da máquina hospedeira, permitindo melhor desempenho. Além disso, existe uma extensa comunidade adepta ao Docker, que contribui com melhorias constantes ao Docker.

2.4.3 Kubernetes

Kubernetes é uma ferramenta de orquestração de contêineres inicialmente desenvolvida por engenheiros da Google. Kubernetes providencia primitivas tais como: Nodes, Services, Pods, Deployment, Job, Secret, ReplicaSet, StatefulSet, Volume, dentre outras, para que desenvolvedores possam definir Clusters. Kubernetes resolve problemas de estabelecimento de rede entre as máquinas que compõe o Cluster, sejam elas físicas ou virtuais. Kubernetes também resolve problema de replicação de serviços, todavia não garante que os dados das réplicas serão replicados de maneira consistente (KUBERNETES, 2022). Quando se trata de sistemas com replicação, busca-se uma infraestrutura robusta que permite que cada réplica possa ter seus recursos garantidos (STUBBS; MOREIRA; DOOLEY, 2015), baseado em métricas e tolerante a falhas (VAYGHAN *et al.*, 2021).

2.4.3.1 Detalhes da tecnologia Kubernetes

A disponibilidade do Kubernetes foi avaliada por (VAYGHAN *et al.*, 2021) e o Kubernetes provê mecanismos que operam para deixar o Cluster operacional e servindo. A escalabilidade e recuperação de desastres é algo que o Kubernetes prove automatizações pré-configuráveis. A escalabilidade permite expandir ou retrair a quantidade de nós de trabalho no *cluster*, dependendo da carga sob demanda. A recuperação de desastres trata de prover a restauração de *Pods* em casos de falhas. Dentre os vários tipos de componentes do Kubernetes se pode mencionar alguns básicos:

Nodes: São máquinas físicas ou virtuais que têm como principal função executar os *Pods*. Tipicamente um nó em Kubernetes é visto como um servidor. A infraestrutura do Kubernetes segue o esquema Líder-Seguidores, onde o nó líder opera sob os nós seguidores.

Services: Um *Service* define um conjunto de *Pods* que provê serviços variados.

[1] `https://github.com/docker`

Pod: A menor unidade que o orquestrador administra, um *Pod* pode conter múltiplos contêineres em execução, mas tipicamente cada *Pod* contém apenas uma aplicação isolada em contêiner. *Pod* são efêmeros, significa que podem encerrar seu ciclo de vida, se um *Pod* morrer um novo *Pod* será alocado no lugar. Cada *Pod* recebe um endereço IP[2] mas se um *Pod* morrer e for substituído por um novo, o endereço IP permanece o mesmo.

Kubectl: É uma ferramenta de comandos em linha, para ser executado em um *Terminal*. O Kubectl permite executar operações em Kubernetes, mas é preciso um arquivo *YAML* que descreve tudo sobre o *cluster* Kubernetes.

PersistentVolumes: Se trata de um componente que administra a persistência dos dados, podendo ser local ou remoto. O Kubernetes não faz backup dos dados, ficando a cargo do desenvolvedor implementa-lo. Os modos de acesso aos volumes persistentes são: leitura e escrita por um nó único (RWO); somente leitura por vários nós (ROX); leitura escrita por vários nós (RWX).

Kubelet: É um componente básico de cada um Nó do Kubernetes. O *Kubelet* é requisitado pelo Kubernetes do nó controlador para receber informações sobre estatísticas de todos os nós do cluster, com essas essas informações o Kubernetes faz a gestão automática dos Nós e seus componentes internos.

ReplicaSet: Mantêm as réplicas de um *cluster* Kubernetes. É definido com os campos: seletor que especifica como identificar os *Pods*, número de réplicas indicando quantos *Pods*, e um *template* do *Pod* para poder replicar igualmente.

Deployment: O Deployment define como criar e atualizar instâncias do seu aplicativo.

Jobs: é um componente que pode criar um ou mais *Pods*, e executará um procedimento até que os *Jobs* possam terminar. O Kubernetes mantém o número de Jobs que estão finalizando com sucesso e, se esse número chegar até um valor especificado então o Kubernetes vai entender como missão bem sucedida.

[2] IP: *Internet Protocol* se trata de um endereço que identifica um dispositivo na rede.

3 TRABALHOS CORRELATOS

A seguir alguns trabalhos relacionados a ordenação de mensagens em arquiteturas de microsserviços e abordagens em orquestração de contêineres. A pesquisa por trabalhos publicados foi feita de maneira exploratória, visando bases científicas tais como: ACM, IEEE, SBC, dentre outras.

3.1 RELACIONADOS A IMPLEMENTAÇÃO DE ARQUITETURA DE MICROSSERVIÇOS

Trabalhos recentes na literatura têm apresentado diversas soluções envolvendo ordenação de mensagens em arquitetura de microsserviços. Os benefícios de arquiteturas de microsserviços abrangem: modularidade e escalabilidade.

3.1.1 A Kubernetes controller for managing the availability of elastic microservice based stateful applications

Segundo Vayghan *et al.* (2021) a arquitetura de microsserviços vêm ganhando muita popularidade. A arquitetura de microsserviços faz a junção de módulos fracamente acoplados, que podem ser escalados independentemente. Esse trabalho avalia o uso de microsserviços em contêineres orquestrados por Kubernetes, mas o Kubernetes recebeu um incremento de software que visa controlar os estados de alta disponibilidade dos contêineres. Nesse trabalho de Vayghan *et al.* (2021) foram identificados arquiteturas para implantar aplicações em microsserviços com Kubernetes. Esse trabalho também realizou experimentações com o Kubernetes da perspectiva de sua capacidade de disponibilidade. Os resultados dos experimentos mostraram que as ações de reparo do Kubernetes não puderam satisfazer alguns limites de alta disponibilidade, avaliados. Em outros casos o Kubernetes não pode garantir recuperação do serviço. Os autores propuseram um controlador de estados de alta disponibilidade integrado ao Kubernetes, que permite replicação de estados de aplicação e redirecionamento automático de serviço. As avaliações foram feitas sob as perspectivas de disponibilidade e escalabilidade. Os resultados das investigações mostram que a solução pode melhorar o tempo de recuperação de aplicações *stateful* baseadas em microsserviços, em 50%.

Vayghan *et al.* (2021) implementaram uma ferramenta para alta disponibilidade em arquiteturas de microsserviços em *clusters* Kubernetes. O fizeram por recuperar o estado do serviço depois que o *Pod* do Kubernetes foi reparado. Agregaram ao Kubernetes um novo componente chamado Controlador de Estados de Alta Disponibilidade (do Inglês, *High Avaliability State Controller*, abreviado como SC). Conduziram um conjunto de experimentos em arquiteturas de microsserviços em um sistema distribuído

físico. Para cada arquitetura experimentada pelo trabalho, o número de *Pods* implantados é igual a 10 (dez). Em cada conjunto de experimentos se encerra o contêiner da aplicação dos K *Pods* ativos, onde K são inteiros de 1 até 5. Os outros *Pods* ficam vivos para serem avaliados (discutidos em seguida). Em cada rodada dos experimentos são feitas medições de *disponibilidade*, para cada *Pod* que tenha falhado.

Além disso, são comparados com as falhas simultâneas de múltiplos *Pods* afetam métricas de disponibilidade. Medições de disponibilidade são as métricas que avaliam a disponibilidade do serviço de cada *Pod*. As medições na experimentação medem o tempo de reação, tempo de reparo, tempo de restauração e tempo total de indisponibilidade. Tempo de total de indisponibilidade é um período quando a energia suprida ou outro serviço não está mais disponível ou quando um equipamento é encerrado por alguma razão. Outra medição feita nos experimentos é de tempo de escalonamento e se trata da latência desde o momento de envio das requisições, para escalar máquinas, até o momento do último *Pod* implantado.

Uma possível limitação é que o software implementado deve ser compreendido para poder ser utilizado. Um desenvolvedor deverá buscar o Kubernetes modificado ao invés do original e terá que compreender a implementação feita por Vayghan *et al.* (2021).

Vayghan *et al.* (2021) fez contribuições em algoritmos, ou seja programação de software que agrega funcionalidades novas ao Kubernetes. Outra forma de contribuição é como interceptar a *API* do Kubernetes, ou seja, como observar os eventos. Outros benefícios desse artigo são maneiras de configurar o Kubernetes para prover replicação das instâncias da aplicação. Finalmente, se relaciona com o presente trabalho por estas contribuições mencionadas e nas questões de avaliação do sistema em arquiteturas de microsserviços, dando importância também para escalabilidade.

3.1.2 *Incorporating the Raft consensus protocol in containers managed by Kubernetes: an evaluation*

O trabalho de Netto *et al.* (2020) visa implementar uma solução de replicação por máquinas de estados usando Raft como algoritmo de consenso. A sua solução foi construída no topo do orquestrador de contêineres Kubernetes. Esse trabalho optou por utilizar o Raft ao invés do Paxos, pela didática e praticidade.

Para desenvolver a solução os autores escolheram o algoritmo Raft e buscaram um código livre e aberto em linguagem Go. O código foi inserido no contexto de orquestração de contêineres. Esse trabalho visou utilizar o banco de dados Etcd[1]. Os autores estenderam uma biblioteca para poder acrescentar funcionalidades personalizadas ao Kubernetes. Foram adicionados:

- Mecanismo próprio de descoberta de réplicas usando API do Kubernetes.

[1] `https://etcd.io/` para fins práticos e demonstrativos da solução

- Aceitação de requisições por qualquer réplica.

A implementação de Netto *et al.* (2020) permite que as réplicas líder possam receber requisições e assim é possível redirecionar as requisições para o líder. Se o líder falhar outras réplicas podem executar o papel de líder.

Eles criaram um ambiente de experimentação real, para experimentações diversas. Eles organizaram o sistema distribuído físico para que atendessem as necessidades da pesquisa. Os autores compararam 3 cenários, sendo eles:

- A implementação KRaft
- A implementação Raft (exclusivo ao Etcd[2], embutido no Kubernetes)
- Um sistema não replicado.

O número de clientes fazendo requisições simultâneas aos servidores variou de 4 até 64 para fins de experimentação. Para fazer medições constantes em cada instância os autores usaram um software Linux chamado *dstat*. Os resultados desse trabalho focam bastante em latência (em ms) e vazão (em requisições por segundo). Houveram testes com várias quantidades de clientes acessando a plataforma Kubernetes, e além disso foi comparado a solução KRaft, com Raft, e sistema não replicado.

Alguns dos resultados do artigo mostraram que a latência por tempo foi menor do que o sistema replicado, e não há aprofundamento sobre as causas. Outros resultados apontaram que a solução se mostrou limitada na questão de vazão de requisições por tempo. Já outros resultados mostraram análises dos consumos de recursos, a saber, Memória e CPU. Porém não se demonstrou o quanto as máquinas físicas e o Kubernetes consomem de Memória e, CPU. Não foi possível identificar, com clareza, as comparações entre a solução e o Raft do Kubernetes.

O trabalho em questão é ortogonal ao presente trabalho pelos aspectos de métricas de experimentação, latência e vazão. Construção de um ambiente de experimentação físico. Uso de protocolo de consenso visando implementar ordenação de mensagens com algoritmos de consenso. Abordagens em arquiteturas de microsserviços. Uso de Kubernetes para orquestrar contêineres. Isolamento de aspectos de implementação em contêineres.

[2] O Etcd é banco de armazenamento chave-valor distribuído usado para armazenar e gerenciar as informações críticas que os sistemas distribuídos precisam para continuar funcionando.

3.1.3 *Implementação de um interceptador para ordenação de mensagens em arquiteturas baseadas em microsserviços*

Fonseca (2021) propôs uma arquitetura em ambientes de microsserviços para o desacoplamento da lógica de ordenação de mensagens. Um dos objetivos era manter consistência forte em um sistema replicado. Para isto, implementou-se replicação por máquinas de estados usando intercepção de mensagens que chegam ao serviço Hermes. O serviço, baseia-se em padrões de projeto voltados para arquiteturas de microsserviços. O código criado por Fonseca (2021) provê interfaces que uma vez implementadas possibilitam que outros protocolos de consenso sejam incluídos, e também outros protocolos de comunicação.

O *Hermes* foi programado em linguagem Go[3]. O autor utilizou Kubernetes e Docker para criar o sistema de interceptação, pois cada instância do serviço sendo replicado tem uma outra instância do Hermes à frente, interceptando as requisições. Uma vez interceptadas, as mensagens são submetidas ao protocolo de consenso, para ordenação das mensagens. O *Hermes* utiliza o algoritmo Raft para realizar a ordenação.

A avaliação da ferramenta aconteceu através de experimentação no Emulab (WHITE *et al.*, 2003), um ambiente para criação de uma rede física de máquinas. O provisionamento do ambiente foi automatizado com a ferramenta Ansible[4]. Os cenários básicos de avaliação foram: sem replicação, replicação à nível de aplicação e replicação no interceptador.

As comparações mostram que o Hermes satura relativamente mais rápido que o caso sem replicação. Aparentemente os resultados nos mostram que a medida que a vazão média cresce, a latência (ms) tende a acompanhar, porém em alguns pontos a latência tem um comportamento inverso.

Alguns resultados mostraram uma vazão maior para o cenário de replicação a nível de aplicação. No entanto, para replicação a nível de interceptador mostraram uma vazão menor e em um outro resultado avaliando a latência, inverteram-se os papéis. Aparentemente existem limitações com relação a vazão.

A relevância deste trabalho está relacionada ao fato de o trabalho atual estar estendendo as funcionalidades do Hermes, para adicionar uma nova forma de comunicação. É notável o estudo aprofundado do código Hermes, bem como as técnicas, métodos, conceitos, algoritmos por trás do funcionamento do Hermes.

[3] Golang: `https://go.dev/`
[4] Ansible: `https://www.ansible.com/`

4 HERMES

O Hermes (FONSECA, 2021) é um interceptador de mensagens, entre o cliente e servidor. O Hermes é encarregado de ordenar as mensagens usando algoritmos de consenso. Após ordenadas as mensagens, o Hermes entrega as mensagens aos servidores replicados.

O objetivo do Hermes é abstrair os detalhes da ordenação de requisições em sistemas replicados em arquiteturas de microsserviços. Trata-se de um conjunto de interfaces que visam oferecer uma camada de abstração. O Hermes usa o padrão de projetos *proxy*, uma vez que as requisições chegam até o Kubernetes, o serviço do Hermes as intercepta e faz o tratamento de obter consenso entre as réplicas. O cliente envia as requisições para o Hermes, mas isso fica transparente ao usuário. O módulo Hermes utiliza a biblioteca *Hashicorp Raft*[1].

O Hermes tem propósitos de ser desacoplado, escalável e compartimentado em tecnologia de contêiner Docker e orquestrado em Kubernetes. Utilizar o Kubernetes é parte do mecanismo de interceptação e também serve para fazer gestão das réplicas. O sistema usufrui de mecanismos do Kubernetes que isolam as réplicas em seus *Nodes* específicos:

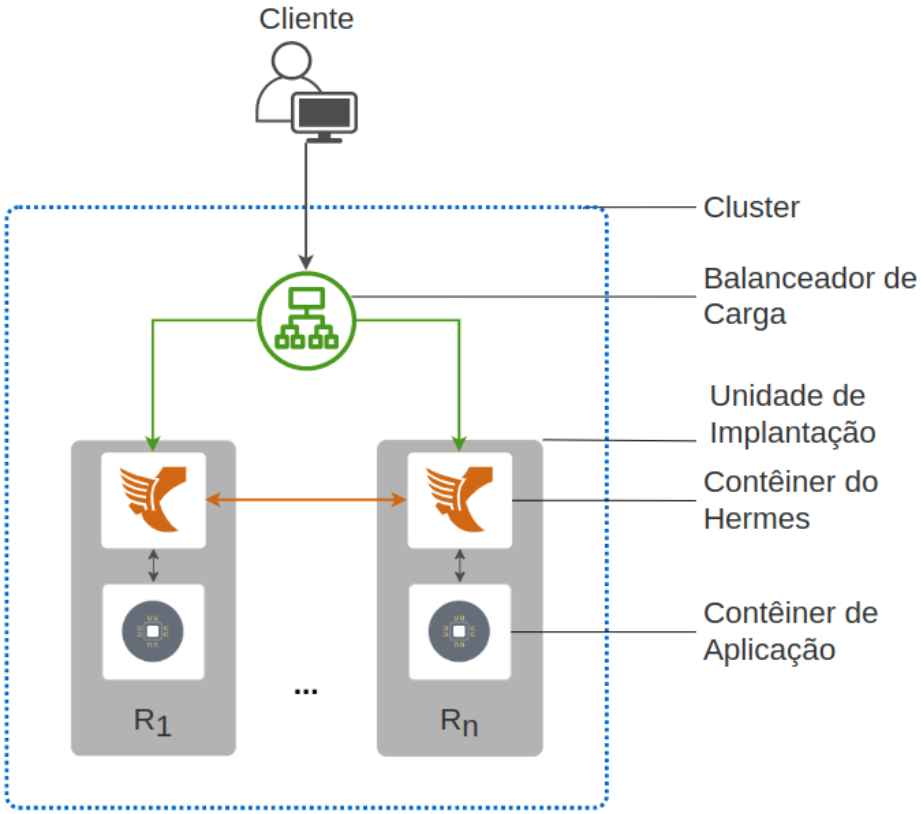
- Afinidade e Anti-Afinidade: Forçar o Kubernetes a inserir uma réplica para cada nó, afastando-as em máquinas físicas, para que cada *Pod* tenha seus recursos garantidos.
- NodeSelector: localizar as réplicas de Hermes com o servidor alvo em Nós especificos com a label *server*. Segregar os clientes dos servidores, em nós.

O interceptador Hermes utiliza padrões de projeto de microsserviços. Os padrões utilizados foram *Proxy* (CHELLIAH *et al.*, 2017) e *Sidecar* (MICROSOFT, 2021). O padrão *Proxy* foi escolhido por fornecer meios lógicos para interceptar mensagens dos clientes e manter um distanciamento das regras de negócio do servidor alvo. O padrão Sidecar foi usado para separar as operações de ordenação de mensagens da aplicação por trás do Hermes, reduzindo sua complexidade interna e deixando a cargo do Hermes para fazer a ordenação de mensagens.

[1] Hashicorp Raft: `https://github.com/hashicorp/raft`

4.1 HERMES EM ARQUITETURA DE ORQUESTRAÇÃO DE CONTÊINERES

Figura 6 – Arquitetura do Hermes



Fonte - Fonseca (2021)

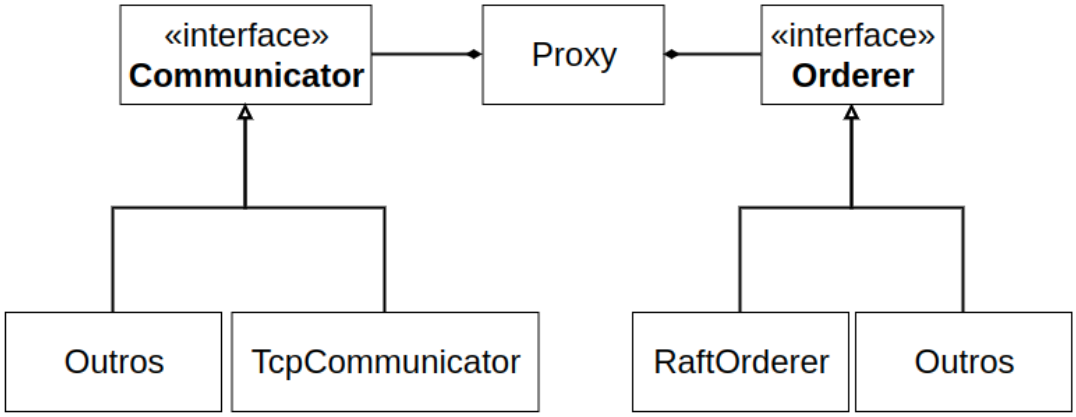
A Figura 6 mostra a arquitetura do Hermes em orquestrador de contêineres. O contêiner do Hermes se localiza à frente do contêiner de aplicação. O bloco cinza se trata da unidade de implantação, onde podem haver as unidades $R_1, R_2, R_3, \ldots, R_n$. A diferença é que quando o cliente envia uma mensagem qualquer para algum serviço, o balanceador de carga irá repassar essa mensagem para um Hermes, esse Hermes fará o papel de interceptar a mensagem e depois irá executar a ordenação de mensagens, uma vez ordenadas as mensagens o Hermes executa a aplicação dessas mensagens nos contêineres da aplicação. Desta maneira se implementa o padrão de projetos Sidecar, pois o Hermes está ao lado do contêiner de aplicação e a comunicação entre os contêineres de Implantação e de Aplicação é feita através de endereço IP e porta.

A aplicação está contida dentro dos *Pods*, e cada *Pod* tem um Hermes e uma aplicação alvo sendo replicada. As requisições chegam para o Hermes líder que repassa a mensagem em bytes para o Raft poder ordenar usando consenso entre os participantes.

4.2 DETALHES DE IMPLEMENTAÇÃO DO HERMES

O desenvolvimento sobre as classes de projeto do Hermes requer que alguns conceitos sejam observados.

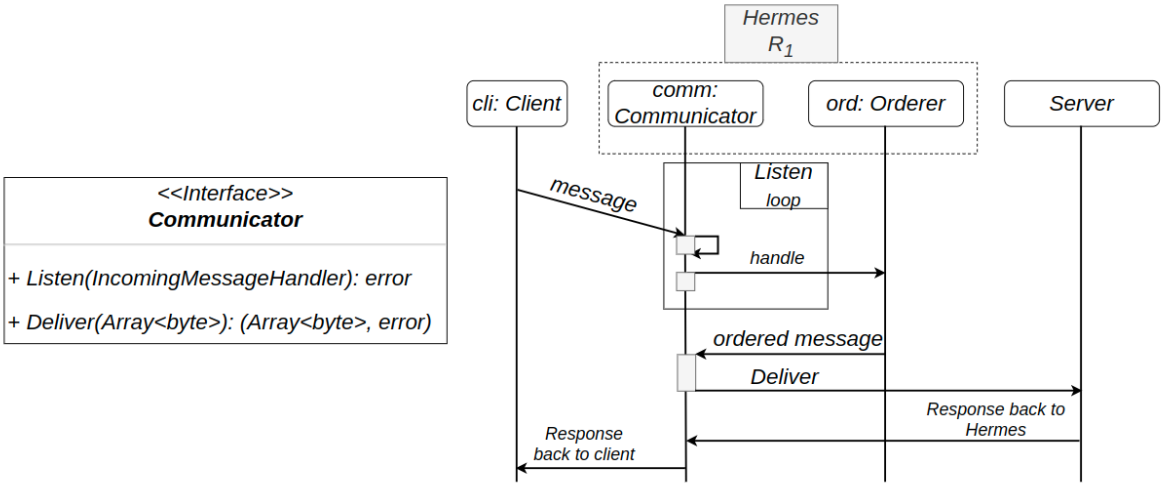
Figura 7 – Relação entre os componentes do interceptador Hermes



Fonte - Adaptado de Fonseca (2021)

A Figura 7 mostra a arquitetura do Hermes em UML. O Hermes foi implementado com padrão de projeto Proxy que por sua vez é composto de 2 objetos, sendo um *comunicador* e um *ordenador*. O *comunicador* implementa a interface *Communicator* e o ordenador implementa a interface *Orderer*.

Figura 8 – Interfaces do código Hermes

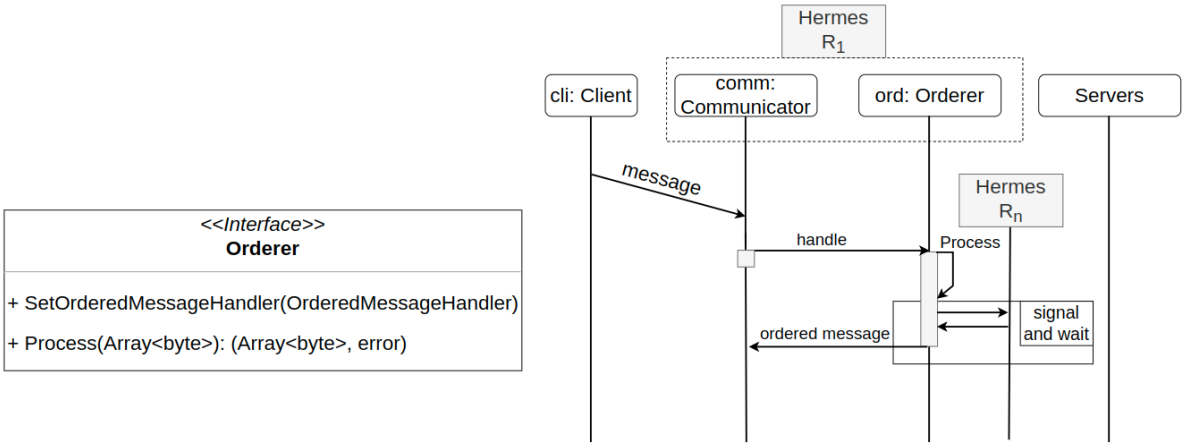


Fonte - Adaptado de Fonseca (2021)

A Figura 8 mostra a interface *Communicator* com os métodos *Listen* e *Deliver* ao lado de um diagrama de mensagens para ilustrar o funcionamento da interface. É possível observar que o método *Listen* permanece escutando mensagens do *Client*.

Assim que uma mensagem é capturada, transforma-se esta mensagem para bytes e se envia para o *Orderer*, através do método *IncomingMessageHandler* que fora previamente configurada. A parte do *Orderer* está omitida no diagrama, porém o *Orderer* irá processar a mensagem, executando um algoritmo específico de ordenação de mensagens *i.e.* Raft ou Paxos. Depois de ordenar a mensagem, o *Orderer* retorna a mensagem em bytes para o método *Deliver*, o *Deliver* irá trabalhar a mensagem e finalmente envia para o servidor de aplicação, porém com a certeza que a mensagem já foi ordenada. Após isto o Hermes ainda pode receber uma resposta do Servidor replicado e assim o Hermes pode repassar a resposta para o *Client*.

Figura 9 – Interfaces do código Hermes



Fonte - Adaptado de Fonseca (2021)

A Figura 9 mostra a interface *Orderer* com os métodos *SetOrdererMessageHandler* e *Process* ao lado de um diagrama de mensagens para ilustrar o funcionamento da interface. Focando no *Orderer*, assim que a mensagem chega para o *Orderer* através do método IncomingMessageHandler, essa mensagem será executada pelo método Process que recebe uma mensagem em *bytes*, depois de executar o algoritmo de ordenação e interagir com as outras réplicas de Hermes então será feita a aplicação da mensagem nos servidor de aplicação.

5 IMPLEMENTAÇÃO DA INTERFACE DE COMUNICAÇÃO NO HERMES

5.1 MUDANÇAS

Foram realizadas mudanças no desenvolvimento na forma de comunicação do Hermes. A mudança permite, o Hermes aceitar requisições HTTP de maneira genérica. Para isso, foi necessário implementar a interface *Communicator*. Durante o desenvolvimento, foi criado um projeto do Hermes com: Docker, Docker Compose e Delve[1]. A implementação tomou o nome de *HttpCommunicator*, seguindo o padrão de nomes previamente implementado.

As requisições HTTP do tipo GET e POST podem ter uma resposta do servidor de aplicação. Esta resposta também foi considerada na implementação do HttpCommunicator, e uma vez que a requisição foi ordenada a requisição pode ser aplicada ao servidor alvo, uma vez que a resposta está pronta ela pode ser repassada ao cliente. O Hermes fará este serviço de ordenação de forma transparente ao usuário.

A intercepção é um trabalho de fatores em conjunto, começando pelo Kubernetes e os arquivos YAML que descrevem o *cluster* Kubernetes. Os arquivos *YAML* têm as descrições de Afinidades, Anti-Afinidades, e *nodeSelector* para atrelar cada *Pod* replicado ao seu nó servidor específico.

5.1.1 Implementação

A classe *HTTPCommunicator* está localizada em *pkg/communication/http.go*. Para implementar a *HTTPCommunicator* foram usadas as bibliotecas *net/http, bufio, bytes, io/ioutil*, dentre outras. A necessidade de estender as bibliotecas de conversão de bytes foi para poder transformar a requisição em bytes e enviar ao *handler* ordenador de mensagens. Uma vez que o Hermes devolve a mensagem ordenada para o *HTTPCommunicator*, é preciso transformar a mensagem de bytes para uma Requisição executável pela biblioteca *net/http*.

[1] Delve é um depurador de código linguagem Go.

Algoritmo 5.1 – Código na versão resumida do HttpCommunicator

```
1  func (comm *HTTPCommunicator) Deliver(/*parameters*/) {
2   // Transforma bytes para request
3   // Altera o host alvo
4   actualRequestRecovered.Header.Set("Host",
5    "http://"+comm.toAddr+actualRequestRecovered.RequestURI)
6   actualRequestRecovered.URL.Host = comm.toAddr
7   actualRequestRecovered.Host = comm.toAddr
8   actualRequestRecovered.RequestURI = ""
9   actualRequestRecovered.URL.Scheme = "http"
10  // Executa request
11 }
12
13 func (comm *HTTPCommunicator) Listen(/*parameters*/)
14  // Transforma request em bytes
15  // Envia ao ordenador
16  handle(bufferedRequestHolder.Bytes())
17 }
```

O Algoritmo 5.1 pode ser entendido da seguinte forma: A linha 14 converte a requisição HTTP para *bytes* (código omitido). A linha 16 faz com que o Hermes acione o mecanismo de ordenação de mensagens, passando os bytes para serem ordenados. Depois que os bytes foram ordenados eles vão eventualmente retornar para o *Deliver* para serem entregues para o servidor da aplicação, para isso a linha 2 converte os bytes em requisição (código omitido). Com a requisição convertida, altera-se o *HOST* alvo nas linhas 4-6. A variável *RequestURI* na linha 8 está sendo transformada para *string* vazia, pois a biblioteca de *net/http* proíbe que a mesma requisição seja usada, desta maneira a biblioteca aceita executar a requisição. Finalmente, a variável *URI.Scheme* na linha 9 está sendo preenchida com o protocolo HTTP.

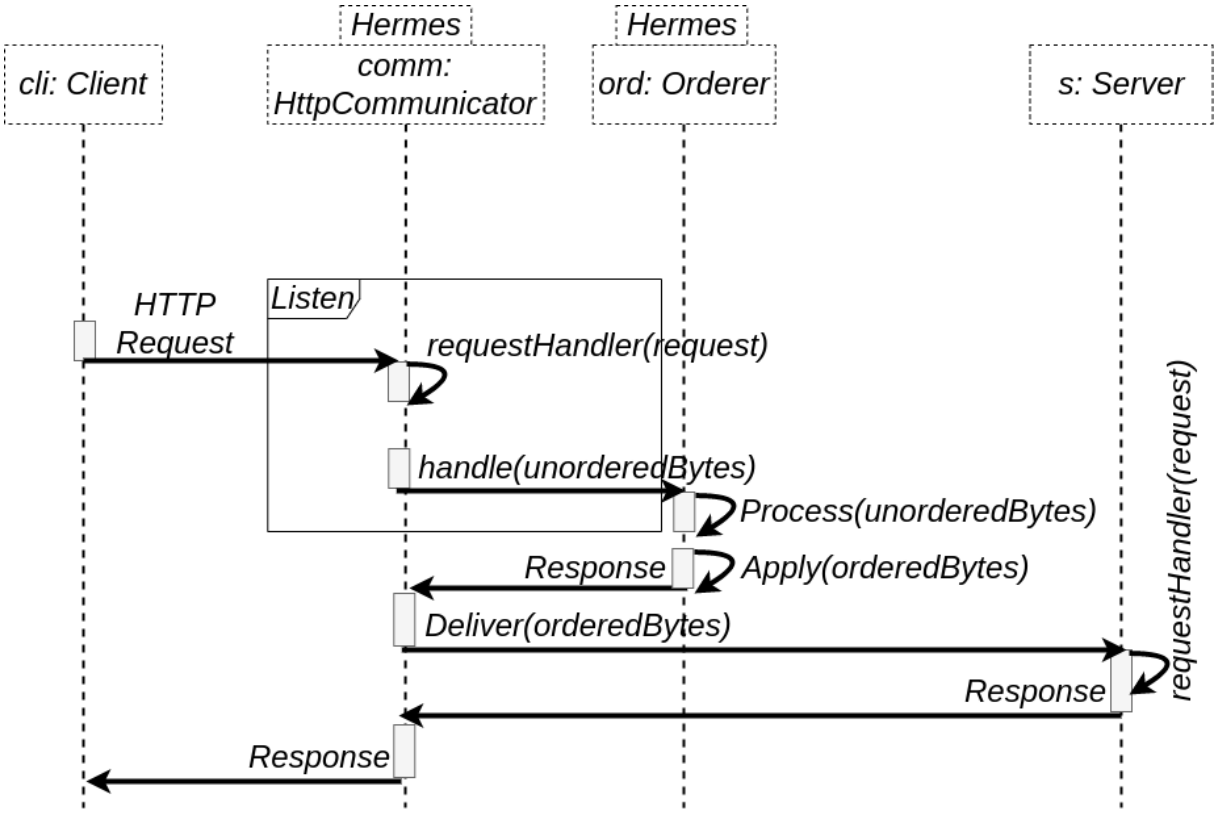
A Figura 10 mostra um diagrama do fluxo de informação passando pelos métodos *Listen* e *Deliver*. Uma vez que uma requisição HTTP qualquer é interceptada pelo Hermes, o evento Listen irá capturar essa requisição e transformar a requisição HTTP em *bytes* e passar ao *handler*. O *handler* irá levar a informação em arranjo de bytes para ser ordenador pelo algoritmo de ordenação. Uma vez que o arranjo de *bytes* está ordenado ele entra no método *Deliver* e portanto eles já estão garantidamente ordenados.

5.1.2 Experimentação

Esta sessão trata como foram realizados as experimentações e, além disso, são mostrados as configurações para os experimentos.

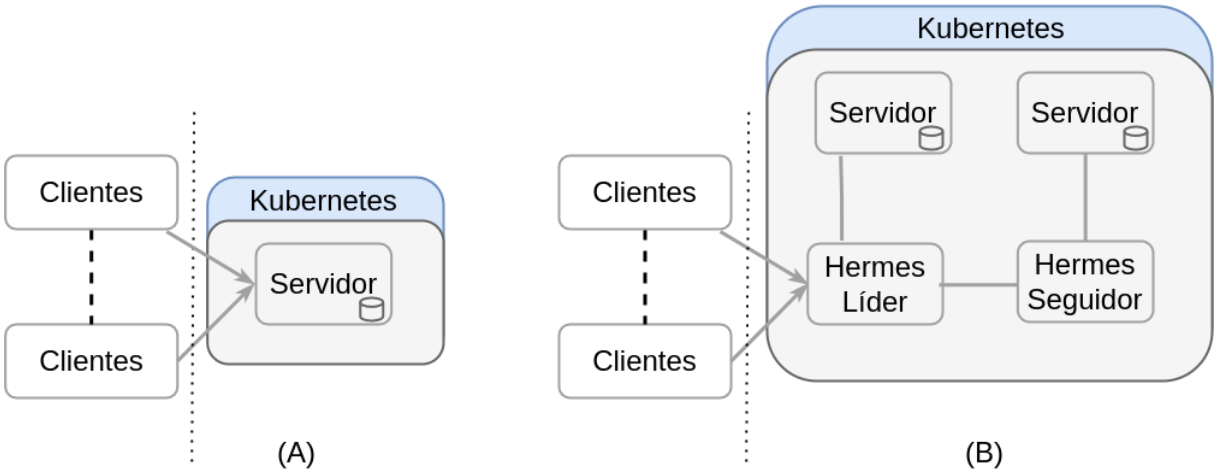
Replicação com ordenação pelo interceptador: A combinação para este cenário utiliza instâncias para gerador de carga, o servidor de armazenamento de *logs* sem consenso e o interceptador Hermes. Este cenário será utilizado para avaliar o custo médio de vazão de requisições HTTP.

Figura 10 – Exemplo resumido dos passos para tratamento da mensagem HTTP que chega ao interceptador Hermes



Fonte - Própria

Figura 11 – Representações das configurações sem replicação (A) e com ordenação respectivamente (B)



Fonte - Adaptado de Fonseca (2021)

A Figura 11 (lado A) mostra a configuração do servidor sem replicação e a Figura 11 (lado B) mostra a configuração do servidor com replicação. Notar que o ordenador de mensagens fica à frente de cada réplica. Estas configurações foram usadas para os experimentos em ambiente real com implantação de orquestrador de

contêineres. Este trabalho inclui duas aplicações que foram implementadas para os experimentos o *HttpLogServer* e o *HttpLogClient*, que serão explicadas a seguir. A aplicação *HttpLogServer* tem basicamente 2 funções:

- Adicionar no fim de um arquivo (*/tmp/logs/operations.txt*) as inserções via requisição POST.
- Iterar sobre as linhas do arquivo */tmp/logs/operations.txt* e retornar a linha requerida via requisição GET.

Foi necessário implementar um servidor HTTP customizado, usando as bibliotecas *http.server* e *sockethandler* para haver liberdade em registrar a contagem de requisições atendidas pelo servidor.

5.1.3 Desenvolvimento

O desenvolvimento do protocolo HTTP foi feito em linguagem Go. O código atual do trabalho pode ser encontrado no Capítulo 7. Caso seja necessário alterar o código fonte, é recomendado alterar o nome do pacote Go, de ‘tonussi/hermes‘ para ‘xyz/hermes‘. Também é recomendado criar um repositório no Github como ‘xyz/hermes‘. Depois de alterar o nome do pacote Go, remova os arquivos *go.mod* e *go.sum* e execute as linhas a seguir:

```
go mod init github.com/xyz/hermes
go mod tidy
go get -u all
```

5.1.4 Detalhes de construção dos contêineres

Para desenvolver no código se recomenda instalar o Docker Compose (define e executa contêineres Docker), o Docker (criador de imagens e contêineres de aplicação), Vscode (editor de código), *make* (programa para executar comandos no Makefile), Makefile é um descritor de comandos úteis a serem executados pelo programa make. Uma vez instalados é possível apenas executar *make build_debug_hermes*, para gerar imagem de contêiner para um Hermes com depurador Delve embutido. Fazendo isto deve ser possível acionar o modo Executar e Depurar selecionando a opção *Debug Hermes (Attach)*, pois há um arquivo *.vscode/launch.json* que integra o Delve com o Docker.

O Depurador indicará qual linha de código está sendo depurada. Como o Go é compilado, não é recomendado alterar o código e re-depurar imediatamente, então basta recompilar o código. O depurador Delve, ajuda a enxergar como o código está se comportando e, quais informações estão passado pelas funções.

Algumas notas importantes são, o arquivo Docker Compose irá gerar: um volume de dados para o BoltDB[2] e um banco de dado usado internamente pelo Raft (interno ao Hermes). Por padrão o Hermes irá escutar o endereço IP:8000 e enviar para o endereço IP:8001.

O servidor da aplicação alvo poderá escutar no endereço IP:8001. A menos que seja necessário alterar os endereços e portas, no caso se o servidor da aplicação precisar escutar outra porta, basta alterar o endereço de envio do Hermes via parâmetros.

[2] BoltDB: `https://github.com/boltdb/bolt`

6 AVALIAÇÃO EXPERIMENTAL

A avaliação foi sobre o Interceptador Hermes versão HTTP e as aplicações: HttpLogClient e HttpLogServer. O HttpLogClient é um gerador de carga escrito em Python e tem a função de estressar o servidor da aplicação, enviando requisições HTTP. O HttpLogClient tem os seguintes parâmetros:

- *address* do tipo *TEXT*: define endereço do servidor;
- *port* do tipo *INTEGER*: define porta do servidor;
- *bytes_size* do tipo *INTEGER*: define o tamanho da carga útil em número de bytes;
- *read_rate* do tipo *INTEGER*: define a taxa de leitura de 0 a 100 por cento;
- *n_processes* do tipo *INTEGER*: define número de processos do cliente;
- *thinking_time* do tipo *FLOAT*: define o tempo de reflexão entre as solicitações;
- *duration* do tipo *FLOAT*: define a duração em segundos;

O HttpLogServer faz duas operações: *get_line(number)* (*GET*) e *append_line(n-bytes-string)* (*POST*). Estas funções são acionadas por requisições HTTP que podem ser realizadas pelos geradores de carga. Para acionar a função *get_line* basta enviar uma requisição *GET* para */line/<number>*. Para acionar a função *append_line* basta enviar uma requisição *POST* para */insert body: "128-byte-string"*. O *HttpLogServer* tem os seguintes parâmetros:

- *address* do tipo *TEXT*: define endereço do servidor;
- *port* do tipo *INTEGER*: define porta do servidor;

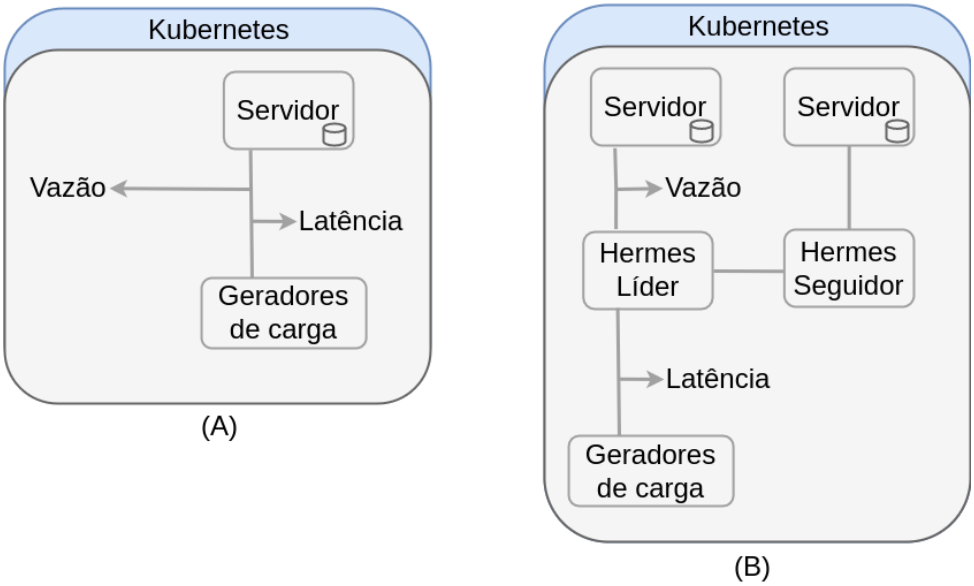
A função *append_line(n-byte-string)* é acionada por requisição *POST* para */insert*, onde o corpo da mensagem é o conteúdo de uma *string* de caracteres ASCII, aleatória, e de tamanho parametrizável. Em um primeiro momento foi necessário avaliar se os dados replicados corretamente, depois que foi visto que os *logs* estavam sendo replicados, então começaram os experimentos de Latência (em nanosegundos) e, Vazão (número de requisições por segundo).

A avaliação se deu por examinar os dados brutos, para entender o comportamento do sistema perante algumas poucas cargas. A medida que se foi possível entender como aplicar cargas, foram feitos melhoramentos sobre os códigos do *HttpLogServer* e *HttpLogClient*.

Por exemplo o uso de *Threads* em Python estava impedindo que os experimentos gerassem dados plausíveis, pois em Python não há concorrência real de *Threads*. No entanto, o uso de Processos se apresentou como uma solução alternativa ao uso de *Threads*.

6.1 MEDIÇÕES

Figura 12 – Esquematização de como e onde foram feitas as medições



Fonte - Própria.

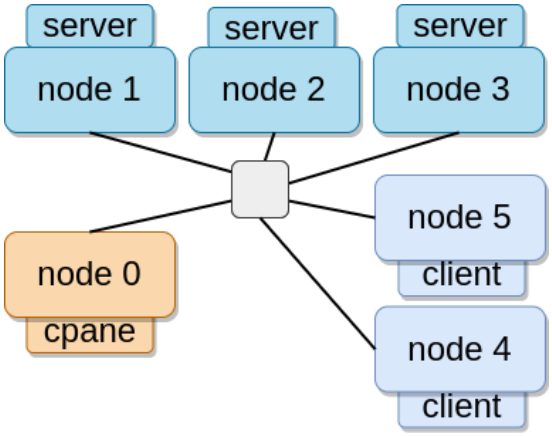
A Figura 12 ilustra em (A) que se trata de medições em um sistema sem replicação e em (B) que se trata de medições em um sistema replicado. Em (B), as entidades com nome *Hermes Seguidor* representam 2 replicações, totalizando 3 réplicas juntamente com o Hermes Líder. A medição da vazão foi realizada como sendo a quantidade de requisições atendidas a cada segundo. A medição da latência foi feita como sendo o quanto tempo demora para executar uma requisição HTTP e obter o retorno. As setas na figura (A) e (B) buscam mostrar onde o fluxo de informação foi capturado. Em (A) e (B) as latências são registradas no programa gerador de carga. Finalmente, em (A) e (B) as vazões são registradas no programa servidor sendo replicado.

6.2 AMBIENTE DE EXPERIMENTAÇÃO

O ambiente de experimentação foi a plataforma Emulab[1] (WHITE *et al.*, 2003). Nesta plataforma foram alocadas 3 máquinas para os servidores de ordenação de mensagens e 2 máquinas para os geradores de carga. A máquina usada para os experimentos foi a d710. A especificações são as seguintes: marca Dell Poweredge R710, processador 2.4 GHz 64-bit Quad Core Xeon E5530 "Nehalem", cache de 8 MB L3, memória de 12 GB 1066 MHz DDR2 RAM (6 módulos de 2GB), discos rígidos de 500GB e 250GB 7200 rpm SATA.

[1] Emulab: `http://docs.emulab.net/`

Figura 13 – Esquematização dos nodos do Emulab



Fonte - Própria.

A Figura 13 representa a configuração de nodos utilizada. A configuração apresentada mostra que foi pensado em deixar os servidores isolados, reduzindo ruídos. Alguns passos podem ser realizados para implantar um *cluster* Kubernetes pronto para experimentação:

- É preciso instalar o Ansible[2] localmente.
- Criar um experimento no Emulab e copiar os *hosts*.
- (Opcional) Listar os hosts no arquivo *ansible/known_hosts*.
- (Opcional) Usar o *script scripts/ssh_hosts.sh* para adicionar as chaves ssh. O script busca as interfaces de IP privada que serão usadas no passo do Kube Flannel. O script vai imprimir no terminal as interfaces e os IPs. Para configurar o Kube Flannel é necessário pegar apenas os IPs privados. IPs começando com 10. são exemplos de IPs privados.
- Configurar os *hosts* do Emulab, em *ansible/hosts*.
- Configurar *ansible_user*, em *ansible/hosts*.
- Configurar o *kubernetes/kube_flannel.yaml* com os nomes das interfaces de rede privada que foram obtidas no passo anterior. Procurar por –IFACE no arquivo *kubernetes/kube_flannel.yaml*.
- Executar o *script scripts/setup-emulab-cluster.sh*, para poder configurar o *cluster* Kubernetes no Emulab. É preciso passar parâmetros para o script poder executar corretamente. Deve ser passado: caminho diretório Ansible; caminho diretório Kubernetes; nome do experimento no Emulab; nome do grupo no Emulab e a quantidade de servidores do tipo *role=server*.
- Instale o *Kubectl* localmente, para poder executar os arquivos de descrição do Kubernetes.

[2] Ansible - `https://www.ansible.com/`

6.3 CARACTERIZAÇÃO DA CARGA DE TRABALHO UTILIZADA

O procedimento para capturar um conjunto de combinações de número de clientes e número de processos por cliente foi através de uma progressão sucessiva de carga, aumento do número de processos, variando o número de máquinas entre 1 e 2. Assim foi possível entender como a aplicação sem replicação estava se comportando com o aumento de carga. A mesma estratégia foi usada para a aplicação com interceptador Hermes à frente da aplicação.

O número total de processos simultâneos ao Hermes variou de 2 à 64. Cada rodada de experimento precisou de mais ou menos 45 segundos de pausa, para que não houvesse ruído em relação ao experimento anterior.

As marcas temporais, de cada medição foram registradas em nanosegundos para haver mais precisão. Gerar carga para apenas escrita via HTTP POST se demonstrou mais consistente do que apenas leitura via HTTP GET. O procedimento para extração dos pontos se dá por coletar durante 90 segundos a cada experimento.

6.4 AVALIAÇÃO DE DESEMPENHO

Esta sessão discute as configurações usadas em ambiente de testes e avalia o custo associado a utilização do interceptador Hermes.

Os experimentos forneceram dados, ponto a ponto. Os dados em determinado momento podem mostrar estagnação de vazão por unidade de tempo em relação à uma latência que só cresce porém com vazão estagnada. Isto pode sugerir a saturação do sistema. A avaliação final compara a saturação de dois cenários diferentes, com replicação e com ordenação via interceptador de mensagens.

Para determinar a saturação se observa a relação entre a latência dos geradores de carga enviando requisições ao Kubernetes. Um sistema saturado mantém conexões aguardando para serem atendidas demorando demais para entregar a resposta solicitada. Os componentes de software utilizados na execução dos experimentos foram:

- *Servidor de armazenamento de log em disco:* Faz o papel de aplicação *stateful* sendo replicada.
- *Gerador de carga:* Faz o papel de cliente.
- *Hermes versão HTTP:* É o Interceptador e ordenador de mensagens HTTP.

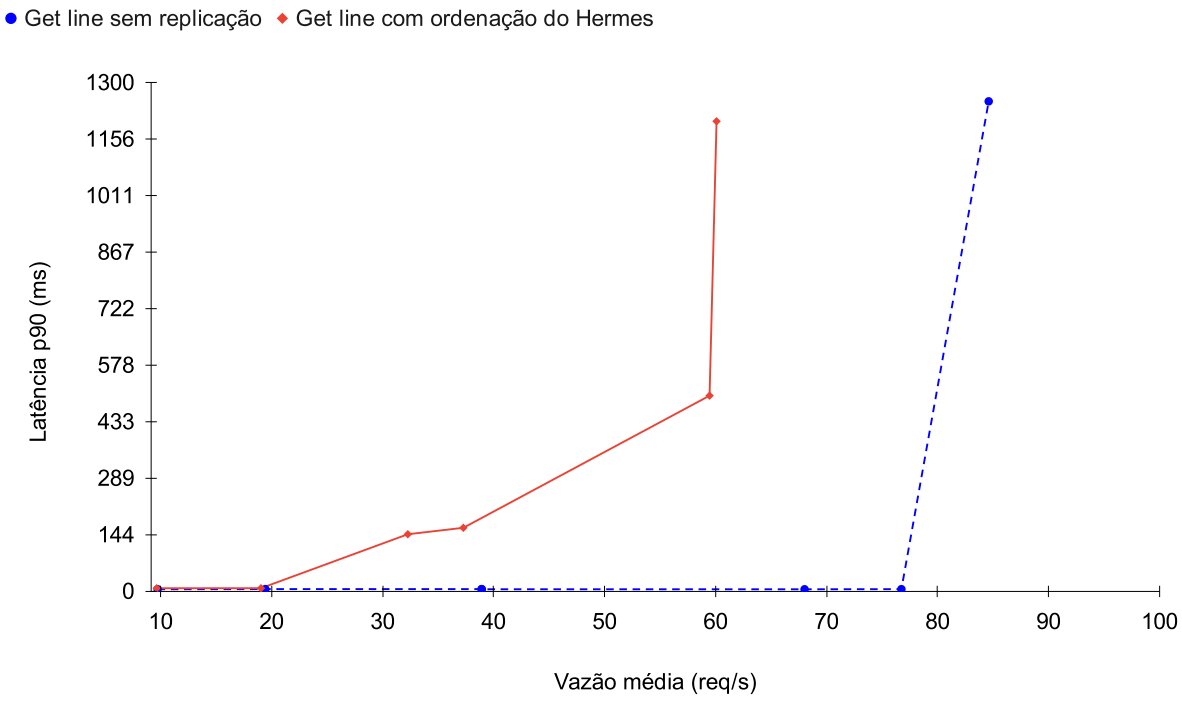
6.4.1 Resultados

As comparações ocorreram entre um sistema sem replicação e um sistema replicado com ordenação de mensagens. As especificações das comparações estão listadas a seguir:

- Requisições GET (100%), acionando *get_line* no servidor, 128 bytes strings, *thinking time* de 200ms.
- Requisições GET (50%) e POST (50%), acionando *get_line* e *append_line* no servidor, 128 bytes strings, *thinking time* de 200ms.
- Requisições POST (100%), acionando *append_line* no servidor, 128 bytes strings, *thinking time* de 200ms.

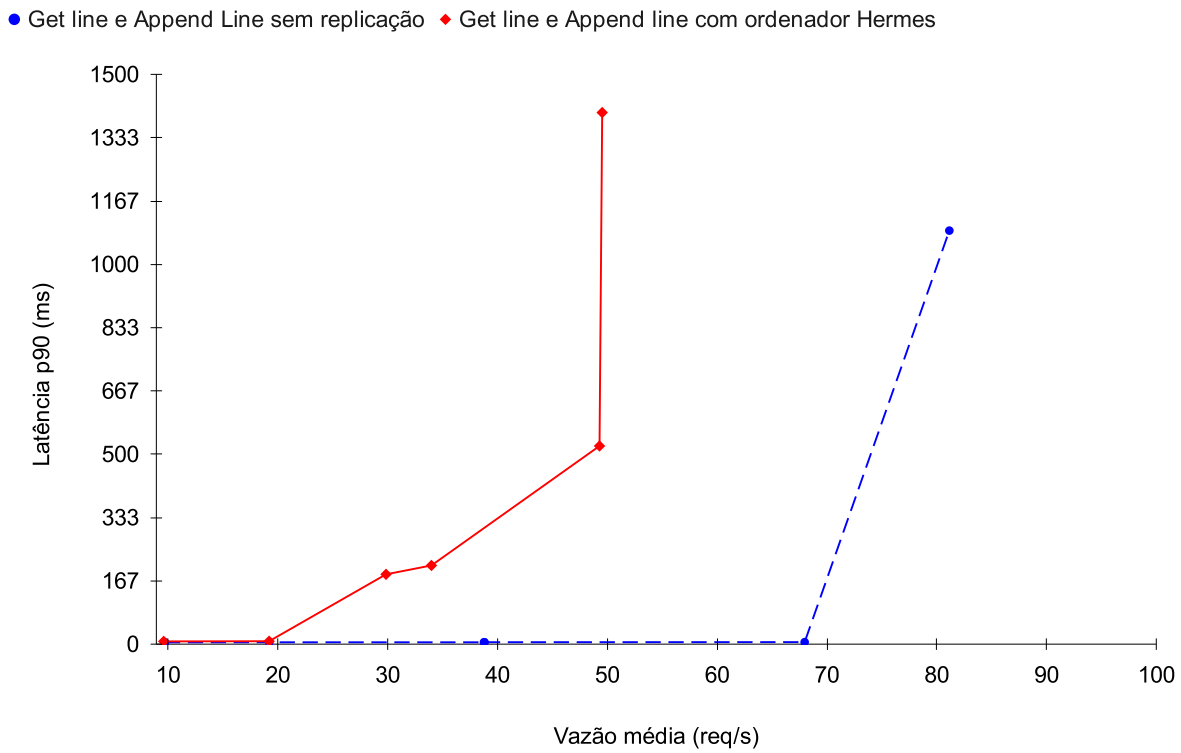
Os experimentos mostram as vazões médias em requisições por segundo no eixo *x* e as latências dadas pelo nonagésimo percentil em milissegundos no eixo *y*. A seguir serão apresentados os gráficos comparativos dos experimentos.

Figura 14 – Requisição GET invocando a função *get_line* no servidor



A Figura 14 mostra que nos pontos de 8-20 req/s o Hermes estava com aproximadamente 7 ms de latência, já no intervalo de 20 até 40 req/s o Hermes obteve latências próximos de 144 ms. O cenário sem replicação obteve latências em torno de 5 ms entre o período de 10 até aproximadamente 68 req/s. O Hermes apresentou consistência até 30 req/s, porém em aproximadamente 60 req/s ocorreu uma estagnação da vazão e crescimento da latência, o mesmo comportamento aconteceu para o sistema sem replicação, porém perto de 80 requisições por segundo. O cenário de 100% requisições GET precisa que exista dados pre-populado com *strings* de 128-bytes para que seja possível obter as linhas, 1000 linhas são pre-populadas e talvez isso faça que com o sistema Hermes obtenha vazão até 60 req/s e então estagne.

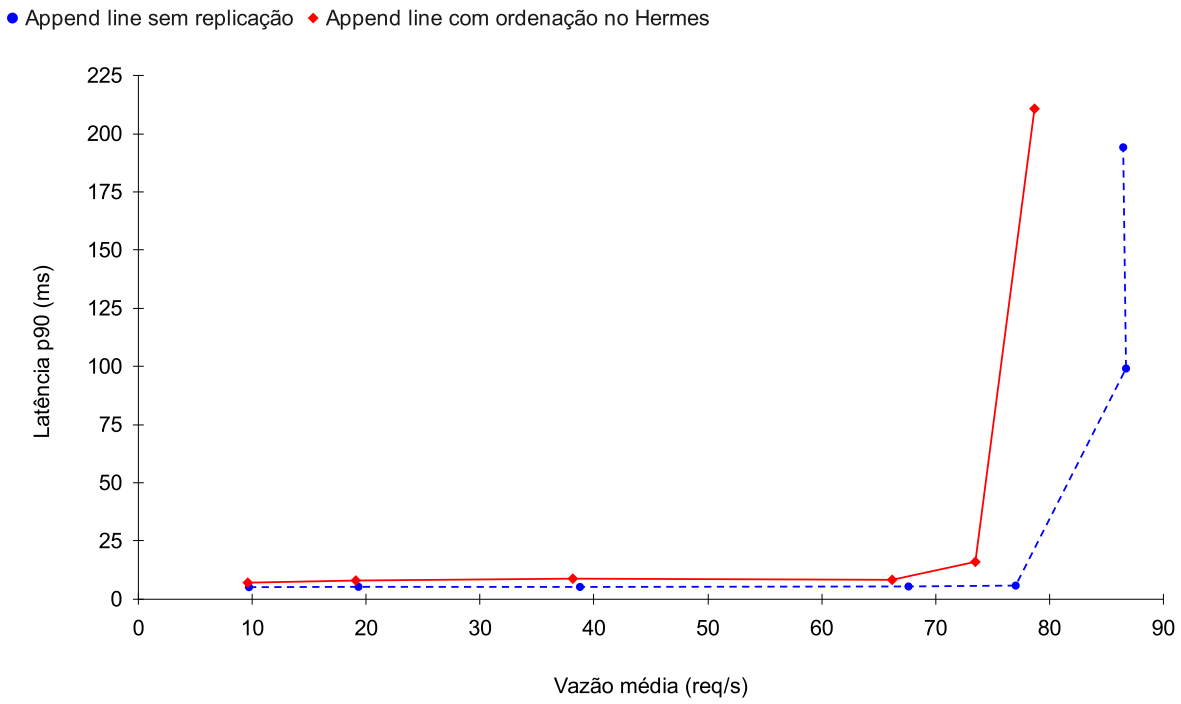
Figura 15 – Requisição GET/POST invocando as funções *get_line* e *append_line* no servidor



A Figura 15 mostra que para o caso de 50% GET e 50% POST faz com que a vazão estagne antes de 60%. Isto pode significar que no caso de 50% GET 50% POST é preciso preencher dados para haver vazão. Notar que em todos os cenários há pre-população de dados. O ponto de saturação do Hermes pode ser observado em aproximadamente 40 req/s, já o ponto de saturação no cenário sem replicação pode ser observado em aproximadamente 80 req/s.

As latências do Hermes se mantiveram aproximadamente 7 ms entre 8 até 20 req/s e a partir de 30 req/s a latência cresceu até aproximadamente 170 ms. Em 60 req/s a latência subiu e estagnou no ponto de 60 req/s. O cenário sem replicação obteve latências de aproximadamente 5 ms desde 10 req/s até aproximadamente 70 req/s e estagnou crescendo latência e obtendo vazões entre 70 e 80 req/s. O ponto de saturação do Hermes pode ser observado em aproximadamente 35 req/s, já o ponto de saturação no cenário sem replicação pode ser observado em aproximadamente 68 req/s.

Figura 16 – Requisição POST invocando a função *append_line* no servidor



A Figura 16 mostra que o Hermes obteve latências de aproximadamente 7 ms entre o período de 10 até aproximadamente 70 req/s. Entre 70 e 80 req/s o Hermes apresentou estagnação da vazão enquanto a latência aumentou para 225 ms. O cenário sem replicação obteve latências de aproximadamente 5 ms entre 10 até aproximadamente 80 req/s. Em 80 req/s começou a estagnar a vazão de requisições, subindo a latência até aproximadamente 200 ms. O ponto de saturação do Hermes pode ser observado em aproximadamente 68 req/s, já o ponto de saturação no cenário sem replicação pode ser observado em aproximadamente 78 req/s.

Em geral, há sobrecarga ao usar o Hermes. A sobrecarga foi baixa no cenário 100% *POST*, que é um comportamento *append-only*, onde apenas se adiciona uma linha no fim do arquivo de (*log*). O custo da latência em ms fica bastante elevado com a presença de comandos *GET*.

7 CONCLUSÃO

A presente monografia apresentou uma forma de comunicação via HTTP para o interceptador e ordenador de mensagens, Hermes. Esta forma de comunicação permite que aplicações que trabalham com HTTP, possam ser replicadas. Os desenvolvedores podem focar na lógica de negócio da sua aplicação ou API, sem se preocuparem com detalhes de ordenação. Este trabalho é também, um estudo sobre: a arquitetura geral do interceptador Hermes; técnicas de ordenação, orquestradores de contêineres, algoritmos de consenso e tópicos relacionados.

O experimentos no Emulab mostraram detalhes sobre a implantação do Hermes, e como operar com ele à frente de uma aplicação feita em Python. Vale lembrar que o uso de orquestrador de contêineres é necessário para tornar operacional o Hermes. A aplicação Python salva as operações dos usuários em *log* no disco. A partir das análises dos experimentos foi possível concluir que o Hermes obteve sobrecarga em relação ao sistema sem replicação.

O uso de técnicas de ordenação de mensagens mostrou que um sistema se mantêm operacional a despeito de uma certa quantidade de falhas das réplicas do serviço, além de oferecer melhor escalabilidade ao serviço replicado. A implementação da interface *communicator* em Go mostrou que protocolos de aplicação (HTTP) podem ser integrados ao sistema de ordenação de mensagens. Os experimentos sem replicação mostram latências menores e mantêm vazões altas por mais pontos de experimentos.

Os experimentos com o ordenador Hermes mostraram latências mais altas em relação ao sistema sem replicação. As leituras em arquivo de uma linha aleatória a cada requisição *GET* fazem com que os gráficos mostrem que o servidor consegue responder em baixas latências até determinada implicação de carga, mas sobe abruptamente em certo ponto. Durante os experimentos, foram feitas diversas tentativas de coleta de pontos e foram criados várias configurações de experimentação, porém os resultados sempre estavam culminando para algo parecido. Contudo, a escrita em arquivo pela requisição POST se demostrou mais previsível nos gráficos, aparentando uma subido gradual da latência enquanto a vazão estagnava. O código usado para este trabalho pode ser encontrado em `https://github.com/tonussi/tcc.git`.

7.1 TRABALHOS FUTUROS

Existe espaço para explorar novas funcionalidades e adicionar integração com novos algoritmos de ordenação de mensagens.

Adição de novos protocolos de comunicação: Por exemplo a implementação do *protocolo UDP* está em aberto para permitir entregas não baseadas em conexão.

Aprimorar o serviço de ordenação proposto em Fonseca (2021): De maneira

geral, pode-se mencionar a implementação de diferentes protocolos de comunicação de ordenação de mensagens, assim como no desenvolvimento de estratégias para aumento da vazão na entrega de requisições às réplicas.

Novos protocolos de consenso no Hermes: Há a possibilidade de inclusão de outros protocolos de consenso, como: Paxos (LAMPORT, 1998), Calvin (THOMSON *et al.*, 2012), CRDT (DEMERS *et al.*, 1987), dentre outros que podem ser encontrados em `https://github.com/heidihoward/distributed-consensus-reading-list`.

Ordenação em lotes: Esta estratégia de ordenação em lotes pode requerer uma terceira fonte de informação, também replicada. Esta fonte guardaria o conteúdo das requisições em bytes com uma chave associada ao lote e uma vez que o ordenador Hermes agrupasse um lote de um tamanho determinado, faria-se então a ordenação de um lote pela chave. Depois de ordenar a chave, recupera-se o lote da fonte de informação e se aplica o lote nas réplicas.

Mudança de líder no sistema Raft do Hermes: Possibilitar que o líder mude, caso ocorra falha do *Pod* que segura o Hermes Líder (nó do Raft atuando como líder). É possível, e aparentemente viável, que essa mudança permita que o sistema de *Load Balancer* do Kubernetes seja integrado ao sistema Hermes. Por enquanto o Hermes líder é fixo, e atua como *Node Port*.

Otimização do atendimento de requisições HTTP: As requisições HTTP do tipo GET, POST, *etc.* podem ter uma resposta do servidor de aplicação. Porém encarregar o Hermes de responder ao cliente pode causar um atraso no processamento de ordenação de mensagens. Sabendo disto, existe espaço para repensar no mecanismo de resposta após aplicar uma mensagem no servidor replicado. Poderia haver um outro serviço que disponibiliza a resposta do servidor em um outro canal de comunicação.

REFERÊNCIAS

AGUILERA, Marcos K; BEN-DAVID, Naama; GUERRAOUI, Rachid; MARATHE, Virendra J; XYGKIS, Athanasios; ZABLOTCHI, Igor. **Microsecond consensus for microsecond applications**. [*S.l.*], 2020. P. 599–616. Disponível em: https://www.usenix.org/conference/osdi20/presentation/aguilera. Acesso em: 16 ago. 2019.

ARAÚJO MACÊDO, Raimundo José de. **Causal order protocols for group communication**. [*S.l.*], 1995. Disponível em: https://www.researchgate.net/publication/228595615_Causal_Order_Protocols_for_Group_Communication. Acesso em: 16 ago. 2019.

BAŠKARADA, Saša; NGUYEN, Vivian; KORONIOS, Andy. **Architecting microservices: Practical opportunities and challenges**. [*S.l.*], 2018. Disponível em: https://doi.org/10.1080/08874417.2018.1520056. Acesso em: 16 ago. 2019.

CARRASCO, Andrés; BLADEL, Brent van; DEMEYER, Serge. **Migrating towards microservices: migration and architecture smells**. [*S.l.*], 2018. P. 1–6. Disponível em: https://doi.org/10.1145/3242163.3242164. Acesso em: 16 ago. 2019.

CASALICCHIO, Emiliano. **Container orchestration: a survey**. [*S.l.*], 2019. P. 221–235. Disponível em: https://link.springer.com/chapter/10.1007/978-3-319-92378-9_14. Acesso em: 16 ago. 2019.

CASON, Daniel. **The role of synchrony on the performance of Paxos: O papel da sincronia no desempenho de Paxos**. Campinas, SP, BR, 2017. Disponível em: http://repositorio.unicamp.br/jspui/handle/REPOSIP/332151. Acesso em: 16 ago. 2019.

CHELLIAH, P.R.; SUBRAMANIAN, H.; MURALI, A.; N, K. **Architectural Patterns: Uncover essential patterns in the most indispensable realm of enterprise architecture**. [*S.l.*], 2017. ISBN 9781787288348. Disponível em: https://www.amazon.com.br/Architectural-Patterns-Pethuru-Raj/dp/1787287491. Acesso em: 16 ago. 2019.

DÉFAGO, Xavier; SCHIPER, André; URBÁN, Péter. **Total order broadcast and multicast algorithms: Taxonomy and survey**. [*S.l.*], 2004. P. 372–421. Disponível em: https://doi.org/10.1145/1041680.1041682. Acesso em: 16 ago. 2019.

DEMERS, Alan; GREENE, Dan; HAUSER, Carl; IRISH, Wes; LARSON, John; SHENKER, Scott; STURGIS, Howard; SWINEHART, Dan; TERRY, Doug. **Epidemic algorithms for replicated database maintenance**. [*S.l.*], 1987. P. 1–12. Disponível em: https://doi.org/10.1145/41840.41841. Acesso em: 16 ago. 2019.

DOCKER. **Docker**. [*S.l.*], jan. 2022. Disponível em: https://docs.docker.com/. Acesso em: 16 ago. 2019.

EKWALL, Richard; SCHIPER, André. **Solving Atomic Broadcast with Indirect Consensus.** [*S.l.*], 2006. P. 156–165. ISBN 0-7695-2607-1. Disponível em: http://doi.ieeecomputersociety.org/10.1109/DSN.2006.65. Acesso em: 16 ago. 2019.

FONSECA, R. T. **Implementação de um interceptador para ordenação de mensagens em arquiteturas baseadas em microsserviços**. Florianópolis, 2021. Acesso em: 16 ago. 2019.

FRITZSCH, Jonas; BOGNER, Justus; ZIMMERMANN, Alfred; WAGNER, Stefan. **From monolith to microservices: A classification of refactoring approaches**. [*S.l.*], 2018. P. 128–141. Disponível em: https://doi.org/10.48550/arXiv.1807.10059. Acesso em: 16 ago. 2019.

GABBRIELLI, Maurizio; GIALLORENZO, Saverio; GUIDI, Claudio; MAURO, Jacopo; MONTESI, Fabrizio. **Self-reconfiguring microservices**. [*S.l.*], 2016. P. 194–210. Disponível em: https://saveriogiallorenzo.com/publications/self-reconfiguring/self-reconfiguring.pdf. Acesso em: 16 ago. 2019.

GHOFRANI, Javad; LÜBKE, Daniel. **Challenges of Microservices Architecture: A Survey on the State of the Practice**. [*S.l.*], 2018. P. 1–8. Disponível em: http://ceur-ws.org/Vol-2072/. Acesso em: 16 ago. 2019.

HADZILACOS, Vassos; TOUEG, Sam. **A Modular Approach to Fault-Tolerant Broadcasts and Related Problems**. USA, 1994. Disponível em: https://dl.acm.org/doi/book/10.5555/866693. Acesso em: 16 ago. 2019.

JAMSHIDI, Pooyan; PAHL, Claus; MENDONÇA, Nabor C.; LEWIS, James; TILKOV, Stefan. **Microservices: The Journey So Far and Challenges Ahead.** [*S.l.*], 2018. P. 24–35. Disponível em: http://doi.ieeecomputersociety.org/10.1109/MS.2018.2141039. Acesso em: 16 ago. 2019.

KUBERNETES. **Kubernetes**. [*S.l.*], jan. 2022. Disponível em: https://kubernetes.io/docs/. Acesso em: 16 ago. 2019.

LAMPORT, Leslie *et al.* **Paxos made simple**. [*S.l.*], 2001. P. 18–25. Disponível em: https://www.microsoft.com/en-us/research/publication/paxos-made-simple/. Acesso em: 16 ago. 2019.

LAMPORT, Leslie. **The Part-Time Parliament**. New York, NY, USA, mai. 1998. P. 133–169. Disponível em: https://doi.org/10.1145/279227.279229. Acesso em: 16 ago. 2019.

LAMPORT, Leslie. **Time, Clocks, and the Ordering of Events in a Distributed System**. [*S.l.*], 1978. Disponível em: https://doi.org/10.1145/359545.359563. Acesso em: 16 ago. 2019.

MICROSOFT. **Sidecar pattern - Cloud Design Patterns | Microsoft Docs**. [*S.l.*], dez. 2021. Disponível em: https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar. Acesso em: 16 ago. 2019.

MILOSEVIC, Zarko; HUTLE, Martin; SCHIPER, Andre. **On the Reduction of Atomic Broadcast to Consensus with Byzantine Faults**. [*S.l.*], 2011. P. 235–244. Disponível em: https://doi.org/10.1109/SRDS.2011.36. Acesso em: 16 ago. 2019.

MOGHADDAM, Fereydoun Farrahi; KANSO, Ali; GHERBI, Abdelouahed. **Simple leaderless consistency protocol**. [*S.l.*], 2016. P. 70–75. Disponível em: https://doi.org/10.1109/IC2EW.2016.23. Acesso em: 16 ago. 2019.

NETTO, Hylson; PEREIRA OLIVEIRA, Caio; RECH, Luciana de Oliveira; ALCHIERI, Eduardo. **Incorporating the Raft consensus protocol in containers managed by Kubernetes: An evaluation**. [*S.l.*], 2020. P. 433–453. Disponível em: https://doi.org/10.1080/17445760.2019.1608989. Acesso em: 16 ago. 2019.

NGUYEN, Nguyen; KIM, Taehong. **Toward highly scalable load balancing in Kubernetes clusters**. [*S.l.*], 2020. P. 78–83. Disponível em: http://dx.doi.org/10.1109/MCOM.001.1900660. Acesso em: 16 ago. 2019.

OLIVEIRA, Caio; LUNG, Lau Cheuk; NETTO, Hylson; RECH, Luciana. **Evaluating Raft in Docker on Kubernetes**. [*S.l.*], 2016. P. 123–130. Disponível em: https://link.springer.com/chapter/10.1007/978-3-319-48944-5_12. Acesso em: 16 ago. 2019.

ONGARO, Diego; OUSTERHOUT, John. **In Search of an Understandable Consensus Algorithm**. Philadelphia, PA, jun. 2014. P. 305–319. ISBN 978-1-931971-10-2. Disponível em: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro. Acesso em: 16 ago. 2019.

STUBBS, Joe; MOREIRA, Walter; DOOLEY, R. **Distributed Systems of Microservices Using Docker and Serfnode**. [*S.l.*], 2015. P. 34–39. Disponível em: https://ieeexplore.ieee.org/document/7217926. Acesso em: 16 ago. 2019.

TAI, Stefan. **Continuous, trustless, and fair: Changing priorities in services computing**. [*S.l.*], 2016. P. 205–210. Disponível em: https://doi.org/10.1007/978-3-319-72125-5_16. Acesso em: 16 ago. 2019.

TERRA, Acacia; CAMARGO, Edson; JR., Elias Duarte. **A Caminho de Uma Alternativa Hierárquica para Implementação do Algoritmo de Consenso Paxos**. Rio de Janeiro, 2020. P. 15–28. Disponível em: https://sol.sbc.org.br/index.php/wtf/article/view/12484. Acesso em: 16 ago. 2019.

THOMSON, Alexander; DIAMOND, Thaddeus; WENG, Shu-Chun; REN, Kun; SHAO, Philip; ABADI, Daniel J. **Calvin: Fast Distributed Transactions for Partitioned Database Systems**. Scottsdale, Arizona, USA, 2012. P. 1–12. (SIGMOD '12). ISBN 9781450312479. Disponível em: https://doi.org/10.1145/2213836.2213838. Acesso em: 16 ago. 2019.

TOFFETTI, Giovanni; BRUNNER, Sandro; BLÖCHLINGER, Martin; DUDOUET, Florian; EDMONDS, Andrew. **An architecture for self-managing microservices**. [*S.l.*], 2015. P. 19–24. Disponível em: https://doi.org/10.1145/2747470.2747474. Acesso em: 16 ago. 2019.

VAYGHAN, Leila Abdollahi; SAIED, Mohamed Aymen; TOEROE, Maria; KHENDEK, Ferhat. **A Kubernetes controller for managing the availability of elastic microservice based stateful applications**. [*S.l.*], 2021. P. 110924. Disponível em: https://doi.org/10.1016/j.jss.2021.110924. Acesso em: 16 ago. 2019.

VERÍSSIMO, Paulo; RODRIGUES, Luís E. T. **Distributed systems for system architects**. [*S.l.*], 2001. P. i–xix, 1–623. (Advances in distributed computing and middelware). ISBN 978-0-7923-7266-0. Disponível em: https://www.amazon.de/gp/reader/B001BV43BE/. Acesso em: 16 ago. 2019.

WHITE, Brian; LEPREAU, Jay; STOLLER, Leigh; RICCI, Robert; GURUPRASAD, Shashi; NEWBOLD, Mac; HIBLER, Mike; BARB, Chad; JOGLEKAR, Abhijeet. **An Integrated Experimental Environment for Distributed Systems and Networks**. New York, NY, USA, dez. 2003. P. 255–270. Disponível em: https://doi.org/10.1145/844128.844152. Acesso em: 16 ago. 2019.

ZDUN, Uwe; WITTERN, Erik; LEITNER, Philipp. **Emerging trends, challenges, and experiences in DevOps and microservice APIs**. [*S.l.*], 2019. P. 87–91. Disponível em: https://ieeexplore.ieee.org/abstract/document/8938118. Acesso em: 16 ago. 2019.

APÊNDICE A – CÓDIGOS DO PROJETO

A.1 BIBLIOTECA

O código fonte pode ser encontrado em: `https://github.com/tonussi/hermes`.

A.2 TESTES DE DESEMPENHO

O código fonte utilizado para os testes de desempenho pode ser encontrado em: `https://github.com/tonussi/stateful-http-api-replication`.

A.3 TRABALHO COMPLETO

O trabalho completo pode ser encontrado em:
`https://github.com/tonussi/tcc`.

ANEXO A – ANEXO - ARTIGO DO PROJETO

Estendendo um Ordenador de Mensagens em Arquitetura de Microsserviços para Comunicação sobre o Protocolo HTTP

Lucas Pagotto Tonussi[1]

[1]Universidade Federal de Santa Catarina

Abstract. *Recently, architectures based on micro-services gained popularity, in part because of the modular programming model, minimum coupling between parts of the system and support on container orchestration platforms, also offering automatic resources management. Message ordering is a strategy that guarantees that all replicas of a distributed system will evolve equally, raising the levels of availability of services and fault tolerance. Aiming applications that use Hypertext Transfer Protocol (HTTP) to operate, this work proposes a implementation of a interface of communication over the HTTP protocol and for a message ordering system. It's known that container orchestrators offers replication automatically, although that replication works, adequately, for stateless applications. This proposal aims to cover cases where the server being replicated is stateful. The objective is to continue the development of a transparent ordering system, for that the present work extends a work previously proposed by the research group, a work called Hermes, which is a service that intercepts messages and take advantage of a container orchestration system. The improvement of that message orderer service counted on a new implementation of the interface that covers communication, the implementation enables the message orderer system to handle HTTP requests. Afterwards, benchmarks of throughput and latency were made. The experiments included two applications: a log application that receives HTTP and saves, in a disk file, the request body and a stress generator application that sends HTTP requests, and can be configured by parameters. The experiments showed that the latencies captured on the stress generators received greater numbers when compared to the non-replicated case, that was expected. The scenario where there is a 100% load of POST requests, the experiment showed a more promising scenario. The case where there is a 100% load of GET requests, the experiment showed better results than the 50% GET 50% POST mixture, because there is a 50% chance of multiple processes inserting more lines into the file. Finally the comparisons between the replicated and non-replicated scenarios showed that the Hermes is able to tolerate failures and replicate HTTP based stateful applications.*

Resumo. *Recentemente, arquiteturas baseadas em microsserviços ganharam popularidade, em parte por causa do modelo de programação modular, acoplamento mínimo entre as partes e o suporte de plataformas de orquestração de contêineres. Ordenação de mensagens é uma estratégia que garante que todas as réplicas evoluam igualmente, aumentando-se os níveis de disponibilidade de serviços. Visando aplicações que usufruem de Hypertext Transfer Protocol (HTTP) para operar, este trabalho propõe uma implementação de interface de comunicação, sobre o protocolo HTTP e para um ordenador de mensagens.*

Sabe-se que orquestradores de contêineres oferecem replicação de forma automática, porém o serviço oferecido por orquestradores garante replicação de aplicações stateless. O objetivo é continuar desenvolvendo um ordenador de mensagens transparente ao usuário, para isto este trabalho estende uma pesquisa iniciada pelo grupo, que propõe o Hermes, um interceptador de mensagens como serviço que usufrui de mecanismos de orquestração de contêineres para prover replicação e tolerância a falhas. O desenvolmento do serviço de ordenação de mensagens contou com a implementação da interface que promove a comunicação no Hermes. A implementação possibilita que o Hermes possa tratar mensagens HTTP. Ao final houve investigação de desempenho da implementação em casos específicos de vazão e latência. Os experimentos incluiram duas aplicações para avaliação de desempenho: uma aplicação de log que recebe requisições HTTP e salva, em arquivo de disco e uma aplicação geradora de carga que envia requisições HTTP, podendo ser configurada por parâmetros. A investigação demonstrou que as latências capturadas nos geradores de carga apresentaram valores maiores para o sistema replicado quando comparado com o caso não-replicado, isto era esperado. O cenário de carga de 100% POST, os experimentos se mostraram mais promissores. O caso onde existe 100% de cargas GET os experimentos se mostraram melhores que no caso híbrido de 50% GET e 50% POST, por causa que existe 50% de chance de múltiplos processos inserirem mais linhas no arquivo de log. Finalmente, as comparações entre os cenários replicados e não-replicado mostraram que o ordenador de mensagens prove tolerância a falhas e replicação ativa de aplicações stateful baseadas em HTTP.

1. Introdução

As arquiteturas de microsserviços têm recebido grande atenção para o desenvolvimento de aplicações distribuídas e vêm sendo amplamente adotadas, especialmente em provedores de computação em nuvem [Aguilera et al. 2020, Netto et al. 2020, Tai 2016, Moghaddam et al. 2016, Nguyen and Kim 2020, Gabbrielli et al. 2016, Toffetti et al. 2015, Oliveira et al. 2016]. Em particular, o desenvolvimento de serviços nessas arquiteturas com suporte de orquestradores de contêineres facilita o gerenciamento de escalabilidade, reaproveitamento de recursos e integração contínua [Ghofrani and Lübke 2018].

2. Motivação

O presente trabalho visa explorar a extensibilidade do Hermes e fazer experimentações em um *cluster* real, estudando a implantação do Hermes. O trabalho de [Fonseca 2021] foi implementado em linguagem Go e o Hermes usa comunicação TCP. Contudo, o presente trabalho se motivou em adicionar a possibilidade do ordenador de mensagens se comunicar via HTTP, ampliando a possibilidade de aplicações se ligarem ao Hermes.

3. Objetivo

Apresentar uma implementação alternativa de comunicação em protocolo HTTP para o interceptador de mensagens Hermes [Fonseca 2021]. Dentre os objetivos específicos, pode-se citar:

- *Meio de comunicação:* Implementar a interface de comunicação do Hermes para possibilitar requisições HTTP.
- *Implementar aplicações de exemplo para testar a integração destas com o serviço de replicação:* Para este propósito, foram desenvolvidas aplicações como *log* em disco.
- *Avaliar a escalabilidade da implementação em protocolo HTTP:* Para isto foram feitas experimentações em ambiente real, extraindo métricas de análise de desempenho.

4. Conceitos Básicos

4.1. Ordem *FIFO*, Causal, Total

Um ordenador de mensagens é um mecanismo vital para promover coerência, em ordenação de eventos entre os participantes de um sistema distribuído e em um sistema replicado, usufruindo-se de replicação ativa, a ordenação de mensagens se torna vital para que o sistema evolua de maneira igual. Existem na literatura várias técnicas de ordenação de mensagens. A seguir são apresentados alguns tipos de ordenação de mensagens.

A ordenação por *First-In First-Out* (FIFO), que significa o primeiro a entrar é o primeiro a sair, estabelece que as mensagens enviadas pelo mesmo participante e entregues para qualquer participante, são entregues na mesma ordem de envio [Veríssimo and Rodrigues 2001].

A ordem causal foi definida por Lamport [Lamport 1978], estabelece algumas premissas: O símbolo m é uma mensagem qualquer que acompanha um ID único e $m_1, m_2, \ldots, m_n \in M$, onde M é um universo de mensagens; $send(m)$ é uma primitiva que executa o envio da mensagem $m \in M$; $p, q, r \in participantes$ onde os participantes são as entidades do sistema, que podem trocar mensagens entre si; $deliver(m)$ é uma primitiva de entrega de mensagens. Uma mensagem m_1 precede logicamente m_2 ***i.e.*** $m_1 \longrightarrow m_2$, sse: m_1 é enviada antes de m_2 pelo mesmo participante **ou** m_1 é entregue para o *enviador* de m_2 antes que ele envie m_2 **ou** existe um m_3 tal que $m_1 \longrightarrow m_3$ e $m_3 \longrightarrow m_2$ [Veríssimo and Rodrigues 2001].

Na ordenação total todos os participantes recebem as mensagens enviadas na mesma ordem. Em termos mais técnicos, defini-se: Quaisquer duas mensagens entregues para quaisquer dois pares de participantes são entregues na mesma ordem para ambos os participantes [Veríssimo and Rodrigues 2001]. O termo difusão em ordem total (do Inglês *Total order broadcast*) é também conhecido como difusão atômica, algumas vezes representado pela primitiva AB-cast. A seguir são apresentada as propriedades necessárias para assegurar ordenação total e os algoritmos para implementação de protocolo AB-cast.

4.2. Consenso

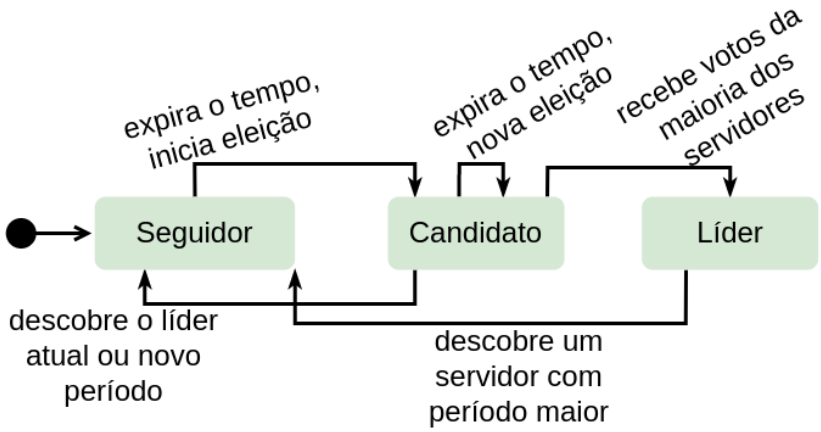
Protocolos de consenso distribuído também podem ser utilizados para implementar entrega totalmente ordenada de requisições [Ekwall and Schiper 2006, Milosevic et al. 2011]. Esses protocolos surgem como uma boa alternativa, visto que a difusão atômica pode gerar uma quantidade muito grande de requisições.

4.3. Raft

O Raft é um algoritmo distribuído e assíncrono de ordenação de eventos [Ongaro and Ousterhout 2014]. Este algorítimo espera que exista um sistema de

replicação de *logs* em cada instância que executa o protocolo, e cada instância do sistema Raft pode estar em um dos seguintes estados: Líder, Candidato, Seguidor, porém só existe um líder por vez e o líder recebe requisições do cliente e pode propor mensagens aos seguidores.

Figure 1. Fases de um serviço Raft



Fonte - Adaptado de [Ongaro and Ousterhout 2014]

A Figura 1 ilustra os estados de cada nó participante e as possíveis transições. Os seguidores reagem às propostas do líder. Se os candidatos não recebem mais sinais do líder, elege-se um novo líder dentre os candidatos. Se algum candidato estiver no estado de seguidor, este deve poder votar por propostas que chegam ou votar para a eleição de novos líderes. É responsabilidade do líder fazer a replicação dos *logs*. Deve haver apenas 1 líder por termo. O líder é responsável pela replicação de *logs*.

4.4. Kubernetes

Kubernetes é uma ferramenta de orquestração de contêineres inicialmente desenvolvida por engenheiros da Google. Kubernetes providencia primitivas tais como: Nodes, Services, Pods, Deployment, Job, Secret, ReplicaSet, StatefulSet, Volume, dentre outras, para que desenvolvedores possam definir Clusters. Kubernetes resolve problemas de estabelecimento de rede entre as máquinas que compõe o Cluster, sejam elas físicas ou virtuais. Kubernetes também resolve problema de replicação de serviços, todavia não garante que os dados das réplicas serão replicados de maneira consistente [Kubernetes 2022]. Quando se trata de sistemas com replicação, busca-se uma infraestrutura robusta que permite que cada réplica possa ter seus recursos garantidos [Stubbs et al. 2015], baseado em métricas e tolerante a falhas [Vayghan et al. 2021].

5. Hermes

O *Hermes* [Fonseca 2021] foi programado em linguagem Go. O autor utilizou Kubernetes e Docker para criar o sistema de interceptação, pois cada instância do serviço sendo replicado tem uma outra instância do Hermes à frente, interceptando as requisições. Uma vez interceptadas, as mensagens são submetidas ao protocolo de consenso, para ordenação das mensagens. O *Hermes* utiliza o algoritmo Raft para realizar a ordenação.

A avaliação da ferramenta aconteceu através de experimentação no Emulab [White et al. 2003], um ambiente para criação de uma rede física de máquinas. O provisionamento do ambiente foi automatizado com a ferramenta Ansible. Os cenários básicos de avaliação foram: sem replicação, replicação à nível de aplicação e replicação no interceptador.

6. Emulab

A plataforma Emulab [White et al. 2003] é um *testbed*, um ambiente de experimentação real. Nesta plataforma é possível criar experimentos, onde o Emulab irá criar a rede de computadores pronta para começar a executar computação paralela.

7. Trabalhos Correlatos

7.1. A Kubernetes controller for managing the availability of elastic microservice based stateful applications

Segundo [Vayghan et al. 2021] a arquitetura de microsserviços vêm ganhando muita popularidade. A arquitetura de microsserviços faz a junção de módulos fracamente acoplados, que podem ser escalados independentemente. Esse trabalho avalia o uso de microsserviços em contêineres orquestrados por Kubernetes, mas o Kubernetes recebeu um incremento de software que visa controlar os estados de alta disponibilidade dos contêineres. Nesse trabalho de [Vayghan et al. 2021] foram identificados arquiteturas para implantar aplicações em microsserviços com Kubernetes. Esse trabalho também realizou experimentações com o Kubernetes da perspectiva de sua capacidade de disponibilidade. Os resultados dos experimentos mostraram que as ações de reparo do Kubernetes não puderam satisfazer alguns limites de alta disponibilidade, avaliados. Em outros casos o Kubernetes não pode garantir recuperação do serviço. Os autores propuseram um controlador de estados de alta disponibilidade integrado ao Kubernetes, que permite replicação de estados de aplicação e redirecionamento automático de serviço. As avaliações foram feitas sob as perspectivas de disponibilidade e escalabilidade. Os resultados das investigações mostram que a solução pode melhorar o tempo de recuperação de aplicações *stateful* baseadas em microsserviços, em 50%.

7.2. *Incorporating the Raft consensus protocol in containers managed by Kubernetes: an evaluation*

O trabalho de [Netto et al. 2020] visa implementar uma solução de replicação por máquinas de estados usando Raft como algoritmo de consenso. A sua solução foi construída no topo do orquestrador de contêineres Kubernetes. Esse trabalho optou por utilizar o Raft ao invés do Paxos, pela didática e praticidade.

Para desenvolver a solução os autores escolheram o algoritmo Raft e buscaram um código livre e aberto em linguagem Go. O código foi inserido no contexto de orquestração de contêineres. Esse trabalho visou utilizar o banco de dados Etcd. Os autores estenderam uma biblioteca para poder acrescentar funcionalidades personalizadas ao Kubernetes. Foram adicionados:

- Mecanismo próprio de descoberta de réplicas usando API do Kubernetes.
- Aceitação de requisições por qualquer réplica.

7.3. *Implementação de um interceptador para ordenação de mensagens em arquiteturas baseadas em microsserviços*

[Fonseca 2021] propôs uma arquitetura em ambientes de microsserviços para o desacoplamento da lógica de ordenação de mensagens. Um dos objetivos era manter consistência forte em um sistema replicado. Para isto, implementou-se replicação por máquinas de estados usando intercepção de mensagens que chegam ao serviço Hermes. O serviço, baseia-se em padrões de projeto voltados para arquiteturas de microsserviços. O código criado por [Fonseca 2021] provê interfaces que uma vez implementadas possibilitam que outros protocolos de consenso sejam incluídos, e também outros protocolos de comunicação.

8. Desenvolvimento

Foram realizadas mudanças no desenvolvimento na forma de comunicação do Hermes. A mudança permite, o Hermes aceitar requisições HTTP de maneira genérica. Para isso, foi necessário implementar a interface *Communicator*. Durante o desenvolvimento, foi criado um projeto do Hermes com: Docker, Docker Compose e Delve[1]. A implementação tomou o nome de *HttpCommunicator*, seguindo o padrão de nomes previamente implementado.

A classe *HTTPCommunicator* está localizada em *pkg/communication/http.go*. Para implementar a *HTTPCommunicator* foram usadas as bibliotecas *net/http, bufio, bytes, io/ioutil*, dentre outras. A necessidade de estender as bibliotecas de conversão de bytes foi para poder transformar a requisição em bytes e enviar ao *handler* ordenador de mensagens. Uma vez que o Hermes devolve a mensagem ordenada para o *HTTPCommunicator*, é preciso transformar a mensagem de bytes para uma Requisição executável pela biblioteca *net/http*.

Algoritmo 1. Código na versão resumida do HttpCommunicator

```
1  func (comm *HTTPCommunicator) Deliver(/*parameters*/) {
2   // Transforma bytes para request
3   // Altera o host alvo
4   actualRequestRecovered.Header.Set("Host",
5    "http://"+comm.toAddr+actualRequestRecovered.RequestURI)
6   actualRequestRecovered.URL.Host = comm.toAddr
7   actualRequestRecovered.Host = comm.toAddr
8   actualRequestRecovered.RequestURI = ""
9   actualRequestRecovered.URL.Scheme = "http"
10  // Executa request
11 }
12
13 func (comm *HTTPCommunicator) Listen(/*parameters*/)
14  // Transforma request em bytes
15  // Envia ao ordenador
16  handle(bufferedRequestHolder.Bytes())
17 }
```

O Algoritmo 1 pode ser entendido da seguinte forma: A linha 14 converte a requisição HTTP para *bytes* (código omitido). A linha 16 faz com que o Hermes acione o mecanismo de ordenação de mensagens, passando os bytes para serem ordenados. Depois que os bytes foram ordenados eles vão eventualmente retornar para o *Deliver* para serem entregues para o servidor da aplicação, para isso a linha 2 converte os bytes em

[1]Delve é um depurador de código linguagem Go.

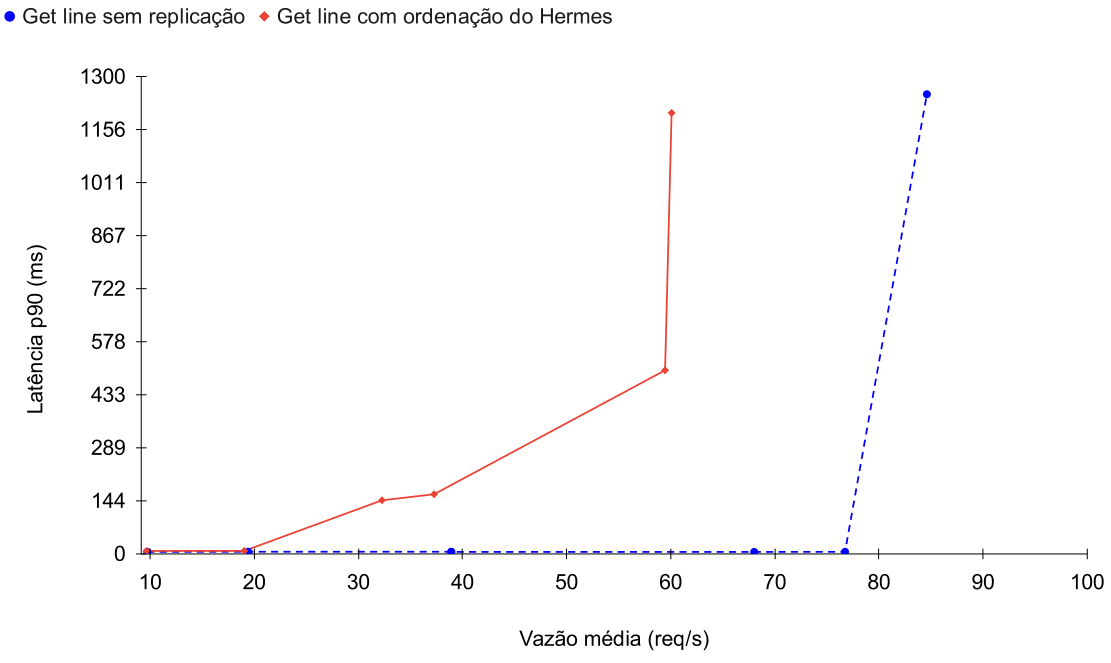
requisição (código omitido). Com a requisição convertida, altera-se o *HOST* alvo nas linhas 4-6. A variável *RequestURI* na linha 8 está sendo transformada para *string* vazia, pois a biblioteca de *net/http* proíbe que a mesma requisição seja usada, desta maneira a biblioteca aceita executar a requisição. Finalmente, a variável *URI.Scheme* na linha 9 está sendo preenchida com o protocolo HTTP. O código completo pode ser encontrado em `https://github.com/tonussi/hermes`.

9. Resultados

O Emulab foi a plataforma utilizada para experimentações e obtenção de resultados. Nesta plataforma foram alocadas 3 máquinas para os servidores de ordenação de mensagens e 2 máquinas para os geradores de carga. A máquina usada para os experimentos foi a d710. A especificações são as seguintes: marca Dell Poweredge R710, processador 2.4 GHz 64-bit Quad Core Xeon E5530 "Nehalem", cache de 8 MB L3, memória de 12 GB 1066 MHz DDR2 RAM (6 módulos de 2GB), discos rígidos de 500GB e 250GB 7200 rpm SATA.

Os experimentos mostram as vazões médias em requisições por segundo no eixo x e as latências dadas pelo nonagésimo percentil em milissegundos no eixo y. A seguir serão apresentados os gráficos comparativos dos experimentos.

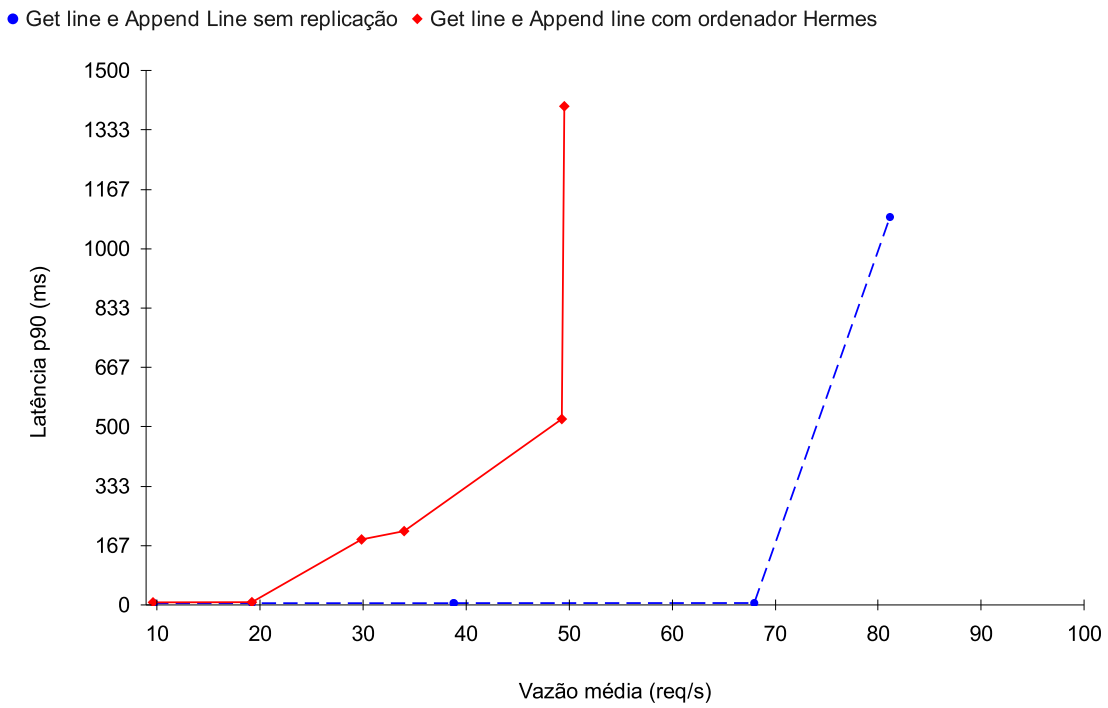
Figure 2. Requisição GET invocando a função *get_line* no servidor



A Figura 2 mostra que nos pontos de 8-20 req/s o Hermes estava com aproximadamente 7 ms de latência, já no intervalo de 20 até 40 req/s o Hermes obteve latências próximos de 144 ms. O cenário sem replicação obteve latências em torno de 5 ms entre o período de 10 até aproximadamente 68 req/s. O Hermes apresentou consistência até 30 req/s, porém em aproximadamente 60 req/s ocorreu uma estagnação da vazão e crescimento da latência, o mesmo comportamento aconteceu para o sistema sem replicação,

porém perto de 80 requisições por segundo. O cenário de 100% requisições GET precisa que exista dados pre-populado com *strings* de 128-bytes para que seja possível obter as linhas, 1000 linhas são pre-populadas e talvez isso faça que com o sistema Hermes obtenha vazão até 60 req/s e então estagne.

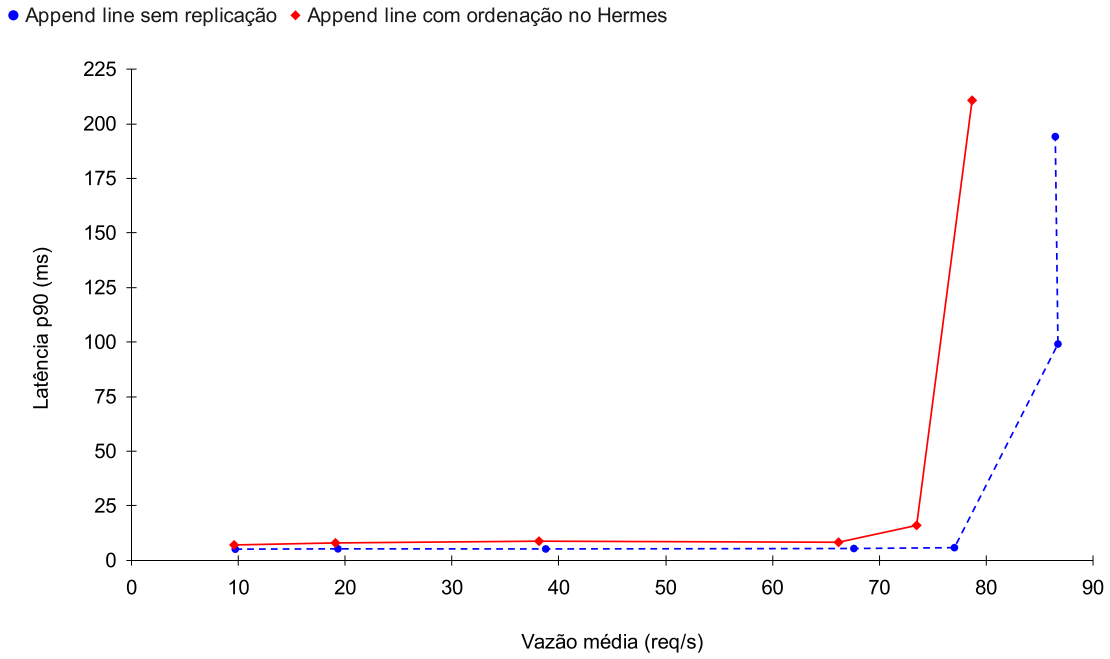
Figure 3. Requisição GET/POST invocando as funções *get_line* e *append_line* no servidor



A Figura 3 mostra que para o caso de 50% GET e 50% POST faz com que a vazão estagne antes de 60%. Isto pode significar que no caso de 50% GET 50% POST é preciso preencher dados para haver vazão. Notar que em todos os cenários há pre-população de dados. O ponto de saturação do Hermes pode ser observado em aproximadamente 40 req/s, já o ponto de saturação no cenário sem replicação pode ser observado em aproximadamente 80 req/s.

As latências do Hermes se mantiveram aproximadamente 7 ms entre 8 até 20 req/s e a partir de 30 req/s a latência cresceu até aproximadamente 170 ms. Em 60 req/s a latência subiu e estagnou no ponto de 60 req/s. O cenário sem replicação obteve latências de aproximadamente 5 ms desde 10 req/s até aproximadamente 70 req/s e estagnou crescendo latência e obtendo vazões entre 70 e 80 req/s. O ponto de saturação do Hermes pode ser observado em aproximadamente 35 req/s, já o ponto de saturação no cenário sem replicação pode ser observado em aproximadamente 68 req/s.

Figure 4. Requisição POST invocando a função *append_line* no servidor



A Figura 4 mostra que o Hermes obteve latências de aproximadamente 7 ms entre o período de 10 até aproximadamente 70 req/s. Entre 70 e 80 req/s o Hermes apresentou estagnação da vazão enquanto a latência aumentou para 225 ms. O cenário sem replicação obteve latências de aproximadamente 5 ms entre 10 até aproximadamente 80 req/s. Em 80 req/s começou a estagnar a vazão de requisições, subindo a latência até aproximadamente 200 ms. O ponto de saturação do Hermes pode ser observado em aproximadamente 68 req/s, já o ponto de saturação no cenário sem replicação pode ser observado em aproximadamente 78 req/s.

Em geral, há sobrecarga ao usar o Hermes. A sobrecarga foi baixa no cenário 100% *POST*, que é um comportamento *append-only*, onde apenas se adiciona uma linha no fim do arquivo de (*log*). O custo da latência em ms fica bastante elevado com a presença de comandos *GET*.

10. Conclusão

O presente artigo apresentou uma forma de comunicação via HTTP para o interceptador e ordenador de mensagens, Hermes. Esta forma de comunicação permite que aplicações que trabalham com HTTP, possam ser replicadas. Os desenvolvedores podem focar na lógica de negócio da sua aplicação ou API, sem se preocuparem com detalhes de ordenação. Este trabalho é também, um estudo sobre: a arquitetura geral do interceptador Hermes; técnicas de ordenação, orquestradores de contêineres, algoritmos de consenso e tópicos relacionados.

Os experimentos com o ordenador Hermes mostraram latências mais altas em relação ao sistema sem replicação. As leituras em arquivo de uma linha aleatória a cada requisição *GET* fazem com que os gráficos mostrem que o servidor consegue responder

em baixas latências até determinada implicação de carga, mas sobe abruptamente em certo ponto. Durante os experimentos, foram feitas diversas tentativas de coleta de pontos e foram criados várias configurações de experimentação, porém os resultados sempre estavam culminando para algo parecido. Contudo, a escrita em arquivo pela requisição POST se demostrou mais previsível nos gráficos, aparentando uma subido gradual da latência enquanto a vazão estagnava.

References

Aguilera, M. K., Ben-David, N., Guerraoui, R., Marathe, V. J., Xygkis, A., and Zablotchi, I. (2020). *Microsecond consensus for microsecond applications*.

Ekwall, R. and Schiper, A. (2006). *Solving Atomic Broadcast with Indirect Consensus.*

Fonseca, R. T. (2021). *Implementação de um interceptador para ordenação de mensagens em arquiteturas baseadas em microsserviços*. Florianópolis.

Gabbrielli, M., Giallorenzo, S., Guidi, C., Mauro, J., and Montesi, F. (2016). *Self-reconfiguring microservices*.

Ghofrani, J. and Lübke, D. (2018). *Challenges of Microservices Architecture: A Survey on the State of the Practice*.

Kubernetes (2022). *Kubernetes*.

Lamport, L. (1978). *Time, Clocks, and the Ordering of Events in a Distributed System*.

Milosevic, Z., Hutle, M., and Schiper, A. (2011). *On the Reduction of Atomic Broadcast to Consensus with Byzantine Faults*.

Moghaddam, F. F., Kanso, A., and Gherbi, A. (2016). *Simple leaderless consistency protocol*. IEEE.

Netto, H., Pereira Oliveira, C., Rech, L. d. O., and Alchieri, E. (2020). *Incorporating the Raft consensus protocol in containers managed by Kubernetes: An evaluation*.

Nguyen, N. and Kim, T. (2020). *Toward highly scalable load balancing in Kubernetes clusters*.

Oliveira, C., Lung, L. C., Netto, H., and Rech, L. (2016). *Evaluating Raft in Docker on Kubernetes*. Springer.

Ongaro, D. and Ousterhout, J. (2014). *In Search of an Understandable Consensus Algorithm*. Philadelphia, PA.

Stubbs, J., Moreira, W., and Dooley, R. (2015). *Distributed Systems of Microservices Using Docker and Serfnode*.

Tai, S. (2016). *Continuous, trustless, and fair: Changing priorities in services computing*. Springer.

Toffetti, G., Brunner, S., Blöchlinger, M., Dudouet, F., and Edmonds, A. (2015). *An architecture for self-managing microservices*.

Vayghan, L. A., Saied, M. A., Toeroe, M., and Khendek, F. (2021). *A Kubernetes controller for managing the availability of elastic microservice based stateful applications*.

Veríssimo, P. and Rodrigues, L. E. T. (2001). *Distributed systems for system architects*.

White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A. (2003). *An Integrated Experimental Environment for Distributed Systems and Networks*. New York, NY, USA.