UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE SISTEMAS DE INFORMAÇÃO

Gabriel Momm Buzzi

Aplicação do *framework* ISS ao protocolo Paxos

Florianópolis
2024

Gabriel Momm Buzzi

Aplicação do *framework* ISS ao protocolo Paxos

Trabalho de conclusão de curso apresentado à Universidade Federal de Santa Catarina como parte dos requisitos necessários para a obtenção do título de Bacharel em Sistemas de Informação.
Orientador: Prof. Odorico Machado Mendizabal, Dr.

Florianópolis
2024

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

BUZZI, Gabriel
Aplicação do framework ISS ao protocolo Paxos / Gabriel BUZZI ; orientador, Odorico Mendizabal, 2024.
101 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Sistemas de Informação, Florianópolis, 2024.

Inclui referências.

1. Sistemas de Informação. 2. Protocolos de consenso. 3. Sistemas distribuídos. 4. Consenso. 5. Algoritmos multi líder. I. Mendizabal, Odorico. II. Universidade Federal de Santa Catarina. Graduação em Sistemas de Informação. III. Título.

Gabriel Momm Buzzi

Aplicação do *framework* ISS ao protocolo Paxos

Trabalho de conclusão de curso apresentado como parte dos requisitos para a obtenção do grau de "Bacharel em Sistemas de Informação".

Florianópolis, Dezembro de 2024

Banca Examinadora:

Prof. Álvaro Junio Pereira Franco, Dr.
Coordenador
Universidade Federal de Santa Catarina – INE

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

Prof.Frank Augusto Siqueira, PhD.
Avaliador(a)
Universidade Federal de Santa Catarina – INE

Prof. Eduardo Camilo Inacio, Dr.
Avaliador(a)
Universidade Federal de Santa Catarina – INE

Este trabalho é dedicado a minha mãe e irmão, que sempre estiveram lá quando eu precisava de ajuda.

RESUMO

Protocolos de consenso são um bloco fundamental para a construção de sistemas distribuídos que garantem a consistência das decisões tomadas por um grupo de processos distribuídos, mesmo em casos de múltiplas falhas. Eles se mostram essenciais para a construção de serviços distribuídos com alta disponibilidade e vazão, como eleição de líderes, replicação de máquinas de estado e bases de dados distribuídas. As variadas necessidades que estes sistemas possuem levaram a criação de diversas variantes de protocolos de consenso, com características que melhor sirvam a cada serviço e situação. Uma grande parte destas variantes, podem traçar sua origem para o Paxos, um dos primeiros protocolos de consenso distribuído e que serviu como base para a criação de novos algoritmos como o Multi-paxos, Fast Paxos, Egalitarian Paxos e WPaxos. Uma destas classes de variantes, os algoritmos multi-líder, são comumente utilizados de forma a atender as demandas crescentes por maior escalabilidade e vazão em sistemas distribuídos. Neste contexto, a presente monografia aplicou o *framework* ISS, capaz de aumentar a escalabilidade e vazão de um protocolo de consenso por meio de uma estratégia multi-líder, ao Paxos, avaliando seu desempenho quanto a vazão, latência e escalabilidade quando comparado a outros protocolos multi-líder. O *framework* ISS, implementado em Go com a ajuda da biblioteca Paxos, apresentou melhor vazão que outros protocolos multi-líder avaliados, no entanto também apresentou elevada latência e potencialmente pior escalabilidade.

Palavras-chave: Sistemas distribuídos, Protocolos de Consenso, Multi-líder, Paxos, Insanely Scalable SMR, ISS.

ABSTRACT

Distributed consensus protocols are one of the fundamental construction blocks of distributed systems. They guarantee the consistency of decisions made by a group of processes, even in cases of multiple failures, being essential to the construction of services with high availability and throughput, such as leader election, state machine replication and distributed databases. The varying needs of these systems has led to the creation of a diverse array of consensus protocol variants, with characteristics that better suit each service and situation. Many of these variants can trace their origins back to Paxos, one of the firsts distributed consensus protocols and base for the creation of new algorithms such as Multi-Paxos, Fast Paxos, Egalitarian Paxos and WPaxos. Some of these variants, classified as multi-leader algorithms, are commonly utilized in order to better meet growing demands for scalability and throughput in distributed systems. As such, this thesis applied the ISS framework, capable of increasing scalability and throughput of consensus protocols by utilizing a multi-leader strategy, to the Paxos protocol, evaluating its performance pertaining to throughput, latency and scalability when compared to other multi-leader protocols. The ISS framework, implemented in Go with the Paxi library, presented a better throughput than other multi-leader protocols that were evaluated, but also presented an elevated latency and potentially worse scalability.

Keywords: Distributed systems. Consensus Protocol, Multi-leader. Paxos. Insanely Scalable SMR. ISS.

LISTA DE FIGURAS

Figura 1 – Estrutura do Paxi (Partes em cinza devem ser implementadas para definir um algoritmo) 21
Figura 2 – Distribuições no Paxi 23
Figura 3 – Reconfiguração no *Cheap* Paxos (Comandos R se referem a reconfigurações e cores diferentes representam diferentes configurações) 25
Figura 4 – Reconfiguração no *Vertical* Paxos (Cores diferentes representam diferentes configurações) 26
Figura 5 – Rodadas de comunicação no Multi-Paxos (assumindo que um líder existe) 27
Figura 6 – Rodadas de comunicação em uma rodada rápida 28
Figura 7 – Exemplo de uma configuração de quorums de grade 4x5 29
Figura 8 – Tipos de falhas possíveis em quorums de grade 30
Figura 9 – Mensagens da fase 2 do *Ring* Paxos (Com anel lógica A1 → A2 → A3 → L) 32
Figura 10 – Paxos clássico (a) e *Ring* Paxos (b,c) 32
Figura 11 – Envio de mensagens no *Pig* Paxos 33
Figura 12 – Partição do *log* no Mencius (cores indicam atribuição à um nodo) . . . 34
Figura 13 – Ordenação de mensagens do *Multi-Ring* Paxos (m=2) 36
Figura 14 – Aprendendo valores no *Multi-Ring* Paxos 36
Figura 15 – Fluxo de mensagens no WPaxos 38
Figura 16 – Fluxo de mensagens no EPaxos 39
Figura 17 – Divisão do *log* em segmentos 42
Figura 18 – Possível erro de ordenação 42
Figura 19 – Possível erro de duplicação 43
Figura 20 – Exemplo de encaminhamento de comandos para *buckets* 43
Figura 21 – Exemplo de encaminhamento de comandos para *buckets* 44
Figura 22 – Distribuição do acesso às chaves 59
Figura 23 – Vazão máxima (3 nodos) 62
Figura 24 – Latência Mediana (3 nodos) 63
Figura 25 – Latência 99 percentil (3 nodos) 64
Figura 26 – Comparação de Vazão (3x6 nodos) 65
Figura 27 – Latência Mediana (6 nodos) 66
Figura 28 – Latência 99 percentil (6 nodos) 67
Figura 29 – Comparação de Vazão (Com e sem otimização) 68
Figura 30 – Latência Mediana (6 nodos otimizado) 69
Figura 31 – Latência 99 percentil (6 nodos otimizado) 70

LISTA DE TABELAS

Tabela 1 – Configurações de cada Nodo 58
Tabela 2 – Parâmetros do gerador de carga 59
Tabela 3 – Parâmetros avaliados para o ISS 60

LISTA DE ALGORITMOS

1 Algoritmo principal do ISS para um nodo p 46
2 Inicialização de época 47
3 Funções auxiliares no ISS 47
4 Obter pedido a partir de grupo de *buckets* 49
5 Co-rotina *handleRequest* 51
6 Co-rotina *runEpoch* 52
7 Gerenciador de *buffer* 53
8 Execução de comandos 54
9 Adaptações ao Paxos 56
10 Adaptações ao Paxos - Detector de falhas 57

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
ISS	Insanely Scalable SMR
LAN	Local Area Network
SB	Sequenced Broadcast
SMR	State Machine Replication
WAN	Wide Area Network

SUMÁRIO

1 INTRODUÇÃO 13
1.1 OBJETIVOS 14
1.1.1 Objetivo Geral 14
1.1.2 Objetivos Específicos 14
1.2 ESTRUTURA DO TRABALHO 14
2 CONCEITOS BÁSICOS 16
2.1 ALGORITMOS DE CONSENSO DISTRIBUÍDO 16
2.2 PAXOS 17
2.3 FERRAMENTAS 19
2.3.1 Paxi 20
2.3.1.1 Gerador de carga 22
3 VARIANTES PAXOS 24
3.1 RECONFIGURAÇÃO 24
3.1.1 *Cheap* Paxos 24
3.1.2 *Vertical* Paxos 25
3.2 TROCA DE MENSAGENS 26
3.2.1 *Fast* Paxos 27
3.2.2 *Flexible* Paxos 28
3.2.3 *Ring* Paxos 30
3.2.4 *Pig* Paxos 31
3.3 ALGORITMOS MULTI-LÍDER 33
3.3.1 Mencius 34
3.3.2 *Multi-Ring* Paxos 35
3.3.3 WAN Paxos (WPaxos) 37
3.3.4 Egalitarian Paxos (EPaxos) 38
4 IMPLEMENTAÇÃO DO PAXOS MULTI-LÍDER 41
4.1 DETALHAMENTO DO ALGORITMO 44
4.2 IMPLEMENTAÇÃO DO ISS NO PAXI 47
4.2.1 Estruturas de apoio 47
4.2.2 Coordenador 50
4.2.3 Adaptações ao protocolo Paxos 55
5 AVALIAÇÃO EXPERIMENTAL 58
5.1 CONFIGURAÇÃO 58
5.2 MÉTRICAS DE AVALIAÇÃO 60
5.3 EXPERIMENTOS COM 3 NODOS 61
5.4 EXPERIMENTOS COM 6 NODOS 63
5.5 OTIMIZAÇÃO DAS MENSAGENS *SKIP* 65

6 **CONCLUSÃO** 71
6.1 TRABALHOS FUTUROS 72
REFERÊNCIAS 73
APÊNDICE A – IMPLEMENTAÇÃO DO ISS NA BIBLIOTECA PAXI 78
ANEXO A – ARTIGO DO PROJETO 80

1 INTRODUÇÃO

Com o constante avanço tecnológico dos dias atuais, cada vez mais a dependência que as pessoas têm em sistemas digitais se fortalece, com aplicações bancárias, de comércio eletrônico e comunicação se tornando praticamente necessárias no dia-a-dia. No entanto, a expansão no uso destes serviços também traz consigo desafios no desenvolvimento destes sistemas críticos, que devem processar grandes quantidades de pedidos simultâneos e manter o sistema funcional, ou ao menos recuperável, mesmo no caso de múltiplas falhas de *hardware*. Para lidar com todos estes pedidos e tolerar falhas sem um grande aumento na latência ou perda de disponibilidade, sistemas distribuídos se mostram essenciais.

Em um sistema distribuído, diversos processos executam simultaneamente um determinado algoritmo de forma a atingir um objetivo em comum por meio da cooperação entre todos os processos (CACHIN; GUERRAOUI; RODRIGUES, 2011). Uma das formas mais comuns de se criar um sistema distribuído é por meio da replicação, onde uma aplicação é inteiramente replicada em diversas máquinas de tal forma que cada um destes processos possa processar um subconjunto dos pedidos dos clientes, aumentando a vazão do sistema. Além disso, no caso de falha de um ou mais processos, os processos corretos podem seguir com suas funções normalmente, evitando a indisponibilização do sistema. De forma a evitar inconsistências entre os múltiplos processos sendo executados simultaneamente, estes devem ser capazes de se comunicar entre si e, mesmo em condições de rede adversas, tomar uma decisão consistente sobre qual o próximo comando a ser executado. Para se chegar em decisões únicas entre múltiplos processos distribuídos, uma das soluções mais comuns é a aplicação de um protocolo de consenso.

Os protocolos de consenso são um bloco fundamental para a construção de sistemas distribuídos (CACHIN; GUERRAOUI; RODRIGUES, 2011). Estes protocolos têm como função permitir a um grupo de processos tomar uma decisão de maneira consistente, ou seja, sem que essa decisão seja perdida ou sobrescrita posteriormente, mesmo com a ocorrência de múltiplas falhas. É a partir de protocolos de consenso que sistemas de alta disponibilidade mais complexos (BURROWS, 2006; HUNT *et al.*, 2010; CORBETT *et al.*, 2012), como replicação ativa, eleição de líderes e sistemas de banco de dados distribuídos podem ser construídos. Como cada um destes sistemas possui necessidades e propriedades específicas, um único algoritmo de consenso não é suficiente para atender todas as demandas de sistemas distribuídos modernos.

Com isso, tomando como base o protocolo de consenso Paxos (LAMPORT, 1998), um dos primeiros criados, novos protocolos de consenso foram sendo desenvolvidos ao longo dos anos, focando em problemas diversos como reconfiguração (LAMPORT; MASSA, 2004; LAMPORT; MALKHI; ZHOU, 2009a), redução de latência (LAMPORT, 2006; LAMPORT; MALKHI; ZHOU, 2009b; CHARAPKO; AILIJIANG; DEMIRBAS, 2021) ou simplicidade de compreensão e implementação (ONGARO; OUSTERHOUT, 2014).

Dentre as variantes do Paxos, existe uma classe específica conhecida como algoritmos multi-líder, que buscam criar sistemas mais escaláveis, ou seja, que possam ser executados em uma maior quantidade de nodos sem uma grande perda de vazão, por meio de uma melhor utilização dos recursos de todos os processos participando do protocolo de consenso.

Algoritmos de consenso distribuído multi-líder se mostram cada vez mais relevantes com o aumento da demanda por sistemas de alta vazão e baixa latência distribuídos geograficamente, como bancos de dados e sistemas de mensagens. Por isto, este trabalho implementa o *framework* ISS (STATHAKOPOULOU; PAVLOVIC; VUKOLIĆ, 2022), capaz de transformar algoritmos de consenso com um único líder em algoritmos de consenso multi-líder, e avalia as propriedades deste quando aplicado ao protocolo Paxos, de forma a determinar se este poderia ser útil para a replicação de sistemas distribuídos que necessitam de alta vazão.

1.1 OBJETIVOS

1.1.1 Objetivo Geral

Este trabalho tem como objetivo aplicar o *framework* ISS ao protocolo Paxos, avaliando o seu desempenho quanto a vazão, latência e escalabilidade em relação a outros algoritmos.

1.1.2 Objetivos Específicos

- Realizar um levantamento de variantes do protocolo de consenso Paxos, de forma a identificar aspectos úteis ao algoritmo implementado.
- Implementar o *framework* ISS.
- Implementar uma variante do protocolo Paxos compatível com o *framework* ISS.
- Realizar uma avaliação experimental do algoritmo implementado em comparação com outros protocolos de consenso voltados a alta vazão e escalabilidade.

1.2 ESTRUTURA DO TRABALHO

O restante deste trabalho está organizado da seguinte forma: o Capítulo 2 apresenta a fundamentação teórica, discutindo as propriedades e usos de algoritmos de consenso, o funcionamento do protocolo Paxos e as ferramentas que foram utilizadas para implementar o *framework* ISS. O Capítulo 3 explora algumas variantes do protocolo Paxos, com foco em variantes que realizem reconfiguração, alterem a estrutura de mensagens de forma a reduzir a latência ou que utilizem estratégias multi-líder. Já o Capítulo 4 apresenta o *framework* ISS, como este funciona e como foi implementado, assim como as modificações realizadas ao protocolo Paxos de forma que este torne-se compatível com o *framework*. No

Capítulo 5, são apresentadas as configurações e resultados dos experimentos realizados e, por fim, o Capítulo 6 apresenta a conclusão do trabalho e possíveis trabalhos futuros.

2 CONCEITOS BÁSICOS

Um dos problemas fundamentais para a construção de sistemas distribuídos é o problema do consenso distribuído, onde deve-se coordenar múltiplos processos distribuídos, cada um propondo um distinto valor v, de forma a garantir que um único valor seja decidido por todos os processos, mesmo no caso de eventual falha de um dos processos (CACHIN; GUERRAOUI; RODRIGUES, 2011). O problema de consenso está presente em diversas classes de aplicações, como bancos de dados distribuídos e serviços de replicação de máquinas de estado, onde comandos devem ser executados na mesma ordem por todos os processos de forma a evitar inconsistências entre diferentes nodos (CACHIN; GUERRAOUI; RODRIGUES, 2011; LAMPORT, 2001). Este capítulo explora o tema dos algoritmos de consenso distribuído, uma classe de algoritmos que soluciona o problema do consenso, e algumas das ferramentas utilizadas para a criação destes.

2.1 ALGORITMOS DE CONSENSO DISTRIBUÍDO

No contexto da computação distribuída, os algoritmos de consenso distribuído são aqueles que permitem a múltiplos processos chegar a uma decisão em comum a partir de um conjunto inicial de valores propostos por estes (VERÍSSIMO; RODRIGUES, 2001; CACHIN; GUERRAOUI; RODRIGUES, 2011). Esta abstração é essencial para a criação de sistemas onde, por motivos de correção ou para tolerar falhas, múltiplos processos devem decidir entre si um único valor, por exemplo, uma ordem de comandos a serem executados, de forma a evitar inconsistências introduzidas por diferentes ordens de execução.

De forma a solucionar o problema do consenso distribuído, os algoritmos de consenso devem garantir as seguintes propriedades (CACHIN; GUERRAOUI; RODRIGUES, 2011):

- **Término:** Todo processo correto eventualmente decide algum valor.
- **Validade:** Se um processo decide o valor v, então v foi proposto por algum processo.
- **Integridade:** Nenhum processo decide duas vezes.
- **Acordo:** Dois processos corretos não decidem valores diferentes.

Para os fins deste trabalho, os protocolos de consenso que serão explorados são aqueles que toleram falhas por colapso e que mantém suas propriedades em ambientes onde o tempo de comunicação é parcialmente síncrono (DWORK; LYNCH; STOCKMEYER, 1988).

Uma falha por colapso ou *crash-fault* é uma falha onde qualquer processo pode, inesperada e indefinidamente, parar por completo sua execução, sem que outros processos detectem a ocorrência da falha (CACHIN; GUERRAOUI; RODRIGUES, 2011; VERÍSSIMO; RODRIGUES, 2001). Isto quer dizer que os algoritmos a serem explorados assumem que processos podem repentinamente parar e nunca se recuperar desta falha.

Já de forma a definirmos um sistema parcialmente síncrono, primeiro é necessário descrever o que são sistemas síncronos e assíncronos. Sistemas síncronos são aqueles onde o tempo máximo entre o envio e recebimento de uma mensagem, assim como o tempo de processamento desta, é definido e conhecido. Já em sistemas assíncronos, não existe um limite máximo para o tempo de comunicação entre dois processos quaisquer (CACHIN; GUERRAOUI; RODRIGUES, 2011; VERÍSSIMO; RODRIGUES, 2001; DWORK; LYNCH; STOCKMEYER, 1988).

Assim, sistemas parcialmente síncronos são aqueles em que o envio de mensagens entre dois processos não apresenta garantia de tempo quanto ao envio e recebimento de mensagens mas que, em certos períodos de execução, a comunicação se comporta dentro dos limites de um sistema síncrono (CACHIN; GUERRAOUI; RODRIGUES, 2011; DWORK; LYNCH; STOCKMEYER, 1988). Este modelo é o que melhor descreve a comunicação entre computadores no mundo real, onde podemos esperar que a maior parte das mensagens sejam entregues dentro de um determinado período de tempo, mas onde não há garantias de que estes limites serão respeitados.

O problema do consenso abrange dificuldades encontradas em sistemas diversos, que vão de sistemas de arquivos distribuídos até serviços de replicação. Assim, ao longo dos anos, vários protocolos, focados em diferentes aspectos do problema, como distintos modelos de falha, condições de comunicação e necessidades de vazão e escalabilidade, foram criados. Alguns exemplos destes protocolos de consenso são o Raft (ONGARO; OUSTERHOUT, 2014), PBFT (CASTRO; LISKOV, 2002), View Stamped Replication (OKI; LISKOV, 1988) e Zab (JUNQUEIRA; REED; SERAFINI, 2011), que já são utilizados comercialmente. Para este trabalho, o protocolo de consenso que será explorado mais a fundo é o Paxos (LAMPORT, 1998).

2.2 PAXOS

O algoritmo Paxos, proposto por Leslie Lamport em 1998, é um protocolo de consenso que coordena múltiplos processos distribuídos de forma a decidir um único valor (LAMPORT, 2001, 1998). No Paxos clássico, cada um dos processos em execução possui pelo menos uma das seguintes funções: *proposer*, *acceptor* ou *learner*.

Um *proposer* propõe valores de forma que um deles seja eventualmente escolhido. Ao sugerir um valor, este processo também envia um número de proposta, que indica o quão recente é a sua proposta, com pedidos antigos sendo descartados por outros processos.

Os *acceptors* são responsáveis por armazenar os dados necessários para que um valor seja decidido, como a indicação do maior número de proposta já recebido e o valor mais recentemente aceito.

Já os *learners* aprendem qual é o valor decidido pelo algoritmo. Este valor decidido é aquele que poderá ser utilizado pela aplicação utilizando este protocolo, podendo ser um comando a ser executado ou uma mensagem recebida, por exemplo.

A execução do algoritmo Paxos em si é composta por duas fases de comunicação distintas: a Fase 1 e a Fase 2, onde a função de cada processo determina as mensagens que ele é responsável por enviar e receber.

Fase 1:

1. Um *proposer* envia uma mensagem `1A` contendo um número de proposta n para todos os *acceptors*.
2. Um *acceptor* ao receber a mensagem `1A`, compara o número de proposta desta com o maior número de proposta conhecido, que é inicialmente nulo.
   - **Caso o número de proposta da mensagem seja maior que o conhecido, ou esta seja a primeira proposta recebida:** O *acceptor* armazena o valor proposto e o número de proposta em sua memória. Depois, ele responde com uma mensagem `1B` para o *proposer*, se comprometendo a não aceitar propostas com número inferior ao recebido. Esta mensagem contém o maior número de proposta m previamente conhecido pelo *acceptor*, assim como o valor v que havia sido proposto em m.
   - **Caso o número de proposta seja menor que o conhecido:** A mensagem é ignorada.

Fase 2:

1. O *proposer*, após receber respostas `1B` de uma maioria de *acceptors*, envia uma mensagem `2A` para todos os *acceptors*. A mensagem `2A` contém o número de proposta n e um valor v. O valor v deve ser o valor presente na mensagem `1B` com maior número de proposta prévio m ou, caso nenhuma proposta prévia tenha sido indicada pelas mensagens, um valor v qualquer que o *proposer* queira propor.
2. Um *acceptor*, ao receber uma mensagem `2A`, compara o número de proposta desta com o maior número de proposta conhecido.
   - **Caso o número de proposta da mensagem seja igual à maior proposta conhecida**: o *acceptor* aceita o valor v proposto. Ao aceitar um valor, o processo então envia uma mensagem `2B`, contendo o valor aceito v e o número de proposta n para um ou mais *learners*.
   - **Caso o número de proposta da mensagem seja menor**: A mensagem é ignorada.
3. Um *learner*, após receber uma mensagem `2B` de uma maioria de *acceptors* contendo o mesmo valor v e número de proposta n, aprende que o valor decidido pelo algoritmo é v.

A partir deste comportamento específico, o protocolo Paxos sempre respeita as propriedades do consenso distribuído de validade, integridade e acordo, sendo que, eventualmente, chega a um término. Implementações do algoritmo Paxos também podem apresentar algumas otimizações que não são requeridas para o funcionamento deste, conforme descrito por Lamport (2001). Dentre estas, temos como as principais:

- Um *acceptor* pode enviar uma mensagem contendo o maior número de proposta n conhecido para o *proposer* ao ignorar suas mensagens `1A` e `2A`, facilitando que o *proposer* envie uma mensagem com um número de proposta maior que n.
- O *acceptor*, ao aceitar uma mensagem `2A`, pode mandar a mensagem `2B` para um único *learner* distinto, que então propaga o conhecimento para os outros *learners* por meio de uma mensagem de *commit*.

No entanto, mesmo com estas otimizações, o algoritmo Paxos clássico, conforme descrito em Lamport (1998), ainda não possui muitas aplicações práticas, pois consegue tomar uma decisão sobre um único valor, enquanto, em geral, é necessário tomar uma sequência potencialmente infinita de decisões. Para se decidir sobre uma sequência de valores, o algoritmo *multidecree* Paxos, também conhecido como multi-Paxos, pode ser utilizado (VAN RENESSE; ALTINBUKEN, 2015; LAMPORT, 2001).

O multi-Paxos baseia-se na execução de múltiplas instâncias do algoritmo Paxos de maneira sequencial, onde a n-ésima instância decide o n-ésimo valor da sequência. Uma otimização frequente para esta variante é permitir que um *proposer* assuma o papel de líder para todas as instâncias subsequentes do algoritmo com o envio de uma única execução da fase 1, de forma que, para as instâncias subsequentes, apenas as mensagens da segunda fase do algoritmo precisem ser enviadas.

A partir do multi-Paxos, surgem novas variantes que buscam otimizar e abordar algumas limitações do algoritmo, como sua dependência em um único líder, utilização de quorums conservativos ou estrutura de mensagem custosa. Algumas destas variantes são discutidas no Capítulo 3.

2.3 FERRAMENTAS

Existem diversas bibliotecas para o desenvolvimento e teste de desempenho entre distintos protocolos de consenso. Estas trazem prontos módulos de comunicação entre processos, aplicações à serem replicadas, estruturas comuns aos protocolos, como *logs*, e, crucialmente, padrões para testar estes algoritmos, de forma que estes possam ser comparados com resultados prévios.

Algumas bibliotecas focam no desenvolvimento de um único algoritmo, como é o caso do URingPaxos (BENZ; BEZERRA, 2013), que implementa o Multi-Ring Paxos

(MARANDI; PRIMI; PEDONE, 2012) em Java, o PhxPaxos (CUI *et al.*, 2018), uma implementação do Paxos (LAMPORT, 1998) em C++, e o BFT-SMaRt (BESSANI; SOUSA; ALCHIERI, 2014; VASSANTLAL *et al.*, 2024), protocolo tolerante a falha bizantinas desenvolvido em Java.

Além das implementações específicas a um único algoritmo, existem bibliotecas que implementam múltiplos algoritmos, levando a uma melhor comparação de desempenho e latência entre estes ao permitir que estes compartilhem uma maior parte de seu código, assim como uma mesma implementação da aplicação e de clientes. Um exemplo desta é o LibPaxos (PRIMI; SCIASCIA, 2013), uma bibioteca desenvolvida em C++ e que traz implementações do algoritmo Paxos (LAMPORT, 1998), RingPaxos (MARANDI; PRIMI; SCHIPER *et al.*, 2010) e FastPaxos (LAMPORT, 2006).

Neste trabalho, a biblioteca utilizada para a implementação do algoritmo desenvolvido foi o Paxi (AILIJIANG; CHARAPKO; DEMIRBAS, 2019). O Paxi foi escolhido por já possuir a implementação de diversos algoritmos de consenso, possibilitando a comparação de desempenho entre estes e o algoritmo desenvolvido. Além disso, por apresentar um projeto modular, este também permite o fácil desenvolvimento de novos algoritmos, construídos a partir de uma arquitetura compartilhada. Por fim, por ser implementado na linguagem de programação Go, ela também facilita a criação de algoritmos com paralelismo e controle de concorrência, já que esta linguagem possui implementações nativas de co-routines (CONWAY, 1963) (chamadas de goroutines) e estruturas nativas para o controle de concorrência.

2.3.1 Paxi

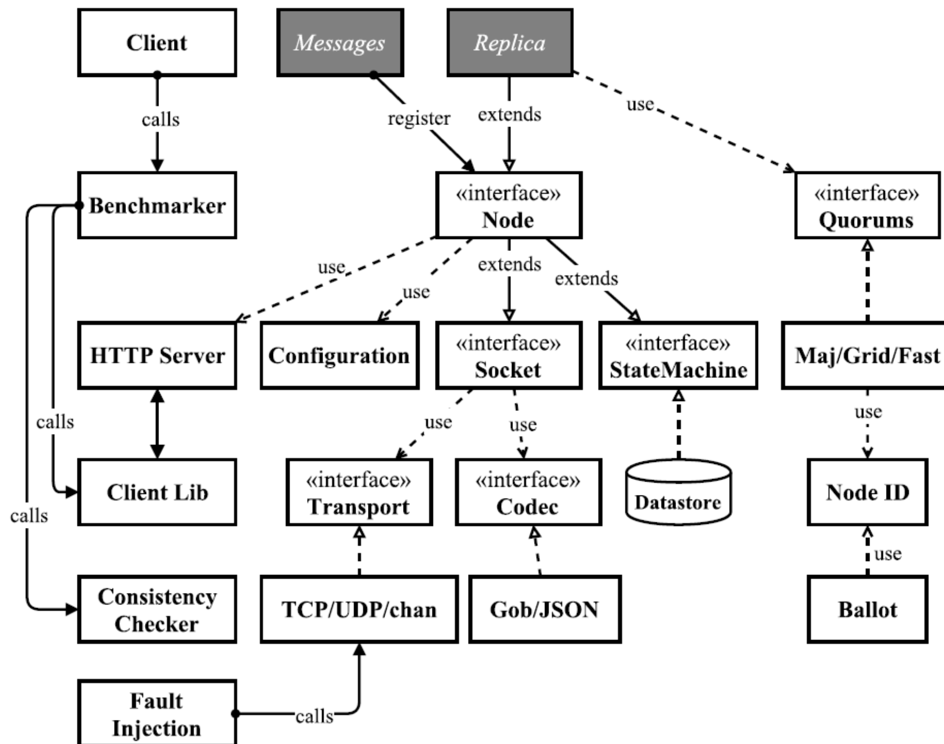
O Paxi (AILIJIANG; CHARAPKO; DEMIRBAS, 2019) é uma biblioteca modular para a implementação de algoritmos de consenso em Go. Os módulos são fracamente acoplados entre si, sendo que a interação entre estes módulos é realizada por meio de uma API bem definida. Desta forma, módulos podem ser facilmente estendidos ou substituídos caso necessário para um determinado protocolo.

Tal estrutura também facilita o rápido desenvolvimento de novos algoritmos, já que apenas alguns componentes deste, como as mensagens e a lógica do algoritmo em si, precisam ser implementados. Para a lógica do algoritmo, é necessário definir como as mensagens e os pedidos dos clientes serão tratados, assim como definir uma função a ser executada para inicializar o algoritmo de consenso.

Os componentes do Paxi podem ser vistos na Figura 1 e podem ser classificados nos seguinte módulos (AILIJIANG; CHARAPKO; DEMIRBAS, 2019):

- **Quorums:** Define uma abstração e implementa diversos modelos de quorums, como maioria simples, quorums rápidos (*fast quorums*) (LAMPORT, 2006), e quorums flexíveis (LAMPORT; MALKHI; ZHOU, 2009b). Isto permite que algoritmos sejam

Figura 1 – Estrutura do Paxi (Partes em cinza devem ser implementadas para definir um algoritmo)



Fonte: Ailijiang, Charapko e Demirbas (2019)

testados sob diferentes modelos de quorums.

- **Rede:** Define uma interface por meio das primitivas *Send*, *Receive*, *Broadcast* e *Multicast* que podem utilizar os protocolos TCP, UDP ou canais da linguagem Go.
- **Base de dados:** Implementa uma base de dados chave-valor, armazenada em memória, e privada para cada Nodo. A base de dados é utilizada para simular uma aplicação sendo replicada pelo algoritmo de consenso.
- **Cliente *RESTful*:** Utiliza uma API *RESTful* para a comunicação entre cliente e servidor. Isto permite a análise de latência e vazão por meio de ferramentas externas, que monitorem a atividade de rede dos clientes e servidores.
- **Gerador de carga (*Benchmarker*):** O componente principal para avaliação dos algoritmos no Paxi, o gerador de carga é o responsável por simular os clientes. Detalhes do gerador de carga são discutidos em mais detalhes na subseção 2.3.1.1.
- **Configurações:** Permite a configuração de diversos parâmetros dos algoritmos, como endereços dos nodos, configuração de quorum e tamanho de *buffers*, e do

gerador de carga, como distribuição na geração de comandos, porcentagem de escritas e número de chaves no banco de dados chave-valor.

2.3.1.1 Gerador de carga

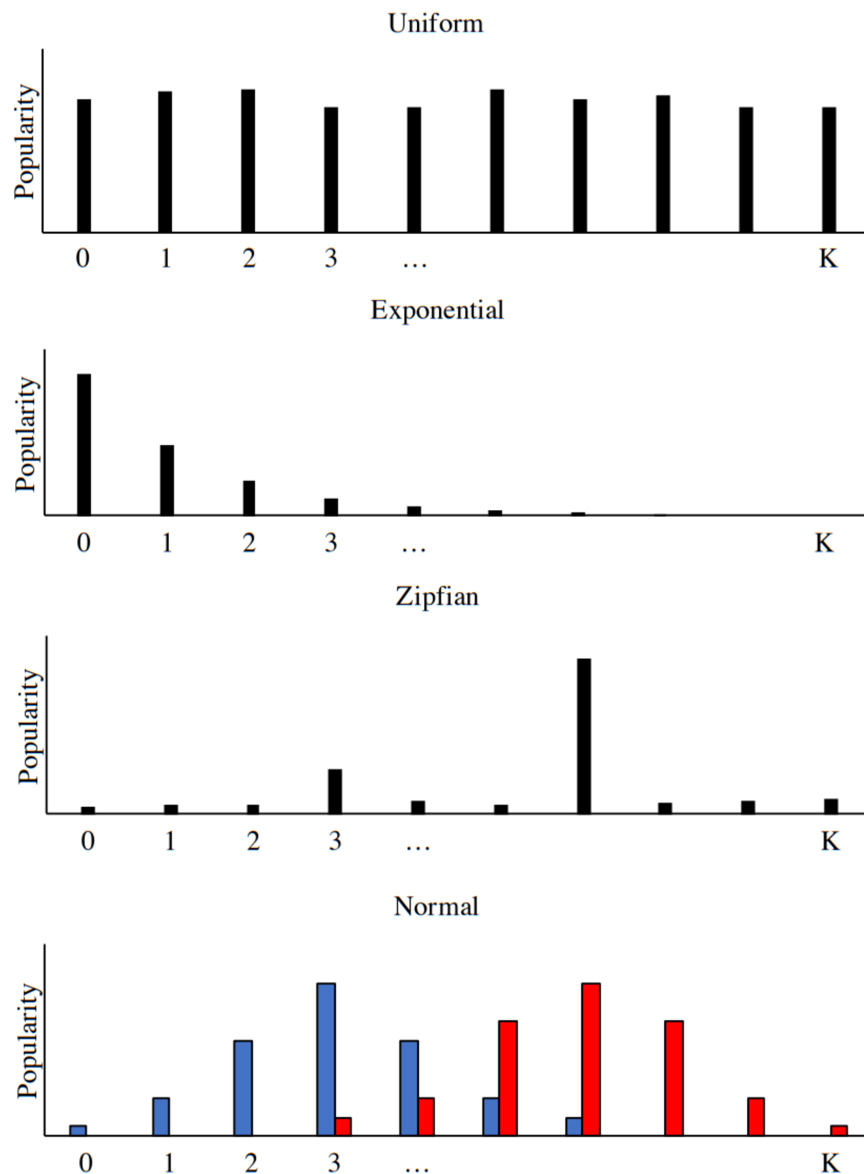
O gerador de carga do Paxi é o responsável por simular o comportamento dos clientes da base de dados chave valor. Um cliente Paxi envia mensagens do tipo *Request*, composta por um identificador do cliente, um número identificador do comando e o comando a ser executado, para um único servidor. O comando contém uma chave do banco de dados e um valor (caso o valor seja nulo, a operação é tratada como uma leitura).

A carga gerada pelo *benchmarker* é dependente dos parâmetros presentes nas configurações, sendo estes:

- **Tempo (T)**: Tempo pelo qual a carga será gerada. Caso seja 0, a carga é gerada com base no parâmetro de número de requisições;
- **Número de Requisições (N)**: Número total de pedidos a serem enviados. Caso seja 0, a carga é gerada com base no parâmetro de tempo de execução;
- **Número de chaves (K)**: Quantidade de chaves distintas a serem simuladas no banco de dados chave valor;
- **Porcentagem de escritas (W)**: Porcentagem de comandos gerados que serão de escrita;
- **Regulador (*Throttle*)**: Quantidade máxima de comandos a serem enviados em um período de 1 segundo;
- **Concorrência (*Concurrency*)**: Quantidade de clientes concorrentes a serem simulados. Isto determina a quantidade máxima de pedidos em trânsito. A quantidade de pedidos enviados ainda é limitada pelo Regulador;
- **Verificador de linearizabilidade**: valor lógico que indica se a verificação de linearizabilidade será executada ao fim da execução;
- **Distribuição (*Distribution*)**: distribuição das chaves a serem acessadas (seja como escrita ou leitura) pelo gerador de carga. Possíveis valores: uniforme (*uniform*), aleatória (*random*), exponencial (*exponential*), zipfiana (*zpfian*) e normal (*normal*). As distribuições podem ser vistas na Figura 2.

Dependendo da distribuição, outros parâmetros também podem ser configurados. A distribuição uniforme não possui outros parâmetros e apenas busca acessar o espaço de

Figura 2 – Distribuições no Paxi



Fonte: Ailijiang, Charapko e Demirbas (2019)

chaves de forma uniforme. Para a distribuição aleatória, é possível definir o menor número de chave a ser acessado e a porcentagem de chaves conflitantes.

Para a configuração de distribuição normal, é possível definir uma distribuição normal ($N(\mu, \sigma^2)$) por meio dos parâmetros μ (Média) e σ (Desvio padrão). Ao manter σ constante e variar o parâmetro μ em cada cliente, também é possível simular clientes com diferentes padrões de acessos as chaves, conforme demonstrado na Figura 2. Além disso, também é possível variar o valor de μ ao longo do tempo, gerando cargas mais dinâmicas e permitindo verificar a adaptabilidade dos protocolos à mudança de padrões de acesso.

3 VARIANTES PAXOS

Como são utilizados para a construção de sistemas distribuídos que possuem diferentes requisitos de latência, vazão e tolerância a falha, não existe um único protocolo de consenso que seja ideal para todas as situações. Assim, ao longo dos anos e tomando como base o algoritmo Paxos (LAMPORT, 1998), novas variantes foram sendo criadas (REGIS; MENDIZABAL, 2022).

Estas variantes aperfeiçoam aspectos específicos do algoritmo original, como sua estrutura de mensagens (LAMPORT, 2006; CHARAPKO; AILIJIANG; DEMIRBAS, 2021) e seu modelo de quorum (LAMPORT; MALKHI; ZHOU, 2009b), ou solucionam problemas práticos, como a reconfiguração (LAMPORT; MASSA, 2004; LAMPORT; MALKHI; ZHOU, 2009a), os custos da comunicação em *Wide-Area Network* (WAN) (AILIJIANG; CHARAPKO; DEMIRBAS; KOSAR, 2020; MORARU; ANDERSEN; KAMINSKY, 2013) e gargalos de execução devido ao *hardware* (MAO; JUNQUEIRA; MARZULLO, 2008; MARANDI; PRIMI; PEDONE, 2012).

Neste capítulo, serão apresentadas algumas variantes do protocolo Paxos, introduzindo protocolos que abordem a reconfiguração, alternativas para redução no número de mensagens enviadas e algoritmos que buscam aumentar a escalabilidade do algoritmo por meio de estratégias multi-líder.

3.1 RECONFIGURAÇÃO

Uma das limitações práticas que o algoritmo Paxos original não aborda em detalhes é a **reconfiguração**. Em um ambiente real, nodos falham e precisam ser substituídos ou novos nodos são adicionados de forma a atender as demandas dinâmicas do sistema. Buscando tratar esta limitação, temos o *Cheap* Paxos (LAMPORT; MASSA, 2004) e o *Vertical* Paxos (LAMPORT; MALKHI; ZHOU, 2009a).

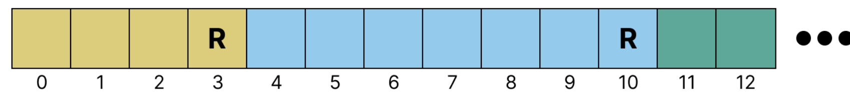
3.1.1 *Cheap* Paxos

O *Cheap* Paxos (LAMPORT; MASSA, 2004) busca permitir a reconfiguração do protocolo de consenso e reduzir os custos com *hardware* sem prejudicar a tolerância a falhas. Para isso, ele apresenta duas ideias principais: a reconfiguração do sistema por meio do protocolo de consenso e a divisão de nodos em principais e auxiliares.

A reconfiguração no *Cheap* Paxos é realizada com a introdução de uma classe especial de comandos, chamados de comandos de reconfiguração, que podem adicionar ou remover um nodo da lista de processos corretos. Estes comandos podem ser propostos por um líder do protocolo de consenso em uma instância do Paxos e adicionados ao *log*, conforme visto na Figura 3, onde, após um comando de reconfiguração ser decidido, a lista de nodos participando do protocolo é alterada. Com esta estratégia, para que um líder do

protocolo Paxos conheça a lista de todos os processos corretos, de forma a estabelecer um quorum na instância i do algoritmo, ele deve conhecer o resultado de todas as instâncias até o ponto $i-1$, tal que ele já tenha executado todos os comandos de reconfiguração previamente estabelecidos.

Figura 3 – Reconfiguração no *Cheap* Paxos (Comandos R se referem a reconfigurações e cores diferentes representam diferentes configurações)



Fonte: De autoria própria.

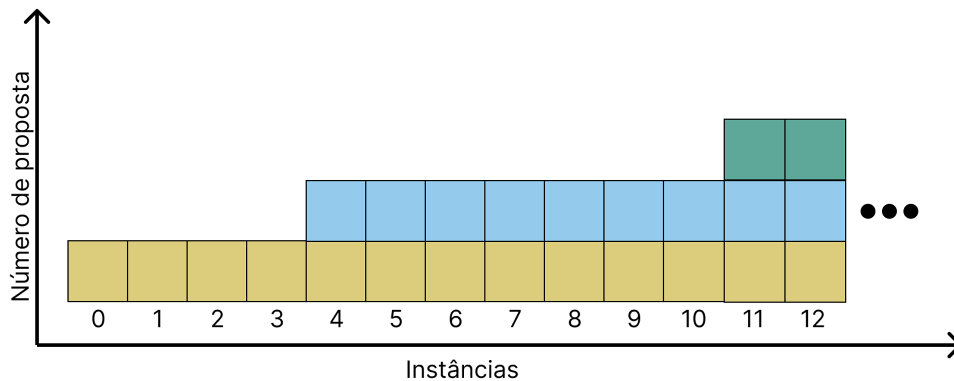
Além disso, o *Cheap* Paxos também busca reduzir custos de *hardware* ao utilizar tanto nodos principais, que executam o protocolo de consenso normalmente, quanto nodos auxiliares, com capacidade de *hardware* reduzida e que permanecem inativos até que uma falha ocorra. Na execução normal do algoritmo, os nodos principais formam um quorum, de forma que os nodos auxiliares ficam inertes. No caso de falha de um dos nodos principais, um quorum diferente, composto de um ou mais nodos auxiliares, é formado. Este quorum é utilizado para completar a execução de instâncias do Paxos que estavam em progresso e, principalmente, remover os nodos defeituosos do protocolo por meio de comandos de reconfiguração. Com isso, os nodos principais novamente formam um quorum e os auxiliares podem voltar a ficar inertes.

3.1.2 *Vertical* Paxos

Outra proposta de reconfiguração é o *Vertical* Paxos (LAMPORT; MALKHI; ZHOU, 2009a), onde, ao invés de realizar decisões de reconfiguração por meio do protocolo de consenso em si, estas são realizadas por um coordenador externo ao protocolo. Nesta variante, quando um nodo é suspeito de falha ou o sistema não pode mais progredir, um dos participantes do protocolo de consenso envia uma mensagem ao coordenador, que então decide em: uma nova configuração, um novo líder para o protocolo e um novo número de proposta. Com isso, diferentemente do *Cheap* Paxos, onde uma configuração está atrelada a uma instância i específica do Paxos, no *Vertical* Paxos cada configuração é associada a um número de proposta, conforme pode ser visto na Figura 4. Nesta, podemos ver operações de reconfiguração ocorrendo durante as instâncias 4 e 11, levando a mudança do número de proposta e, consequentemente, da configuração dos nodos.

Quanto à implementação do coordenador de reconfiguração, este apenas precisa ser capaz de decidir quanto a nova configuração e um líder para o protocolo, sendo que também pode ser implementado como uma versão padrão do protocolo Paxos. Ao coordenador também podem ser atribuídas outras responsabilidades, como acompanhar o número de *log*

Figura 4 – Reconfiguração no *Vertical* Paxos (Cores diferentes representam diferentes configurações)



Fonte: De autoria própria.

mais elevado proposto e outros detalhes do algoritmo de consenso, facilitando o processo de transferência de informação entre diferentes configurações e permitindo uma recuperação mais rápida após falhas.

3.2 TROCA DE MENSAGENS

Um dos principais fatores que determinam o desempenho de um protocolo de consenso é a estrutura de envio e recebimento de mensagens que este utiliza. Quanto a esta estrutura, dois aspectos principais geralmente são levados em consideração: o número de rodadas necessárias para se tomar uma decisão e o número de mensagens a serem enviadas e recebidas por um nodo para cada decisão.

Uma rodada de um protocolo de consenso consiste no envio de mensagens por um conjunto de nodos e eventual recebimento de um subconjunto destas mensagens (LAMPORT, 2000). No algoritmo Paxos original, por exemplo, são necessárias no mínimo quatro rodadas para se atingir o consenso: o envio das mensagens `1A`, envio da resposta `2A`, envio da proposta `2A` e envio da resposta `2B`, sendo que a comunicação com o cliente não é considerada. O número de rodadas afeta o desempenho do algoritmo pois, quanto mais rodadas forem necessárias, maior será o efeito da latência no tempo necessário para tomar uma decisão e maior a utilização da rede.

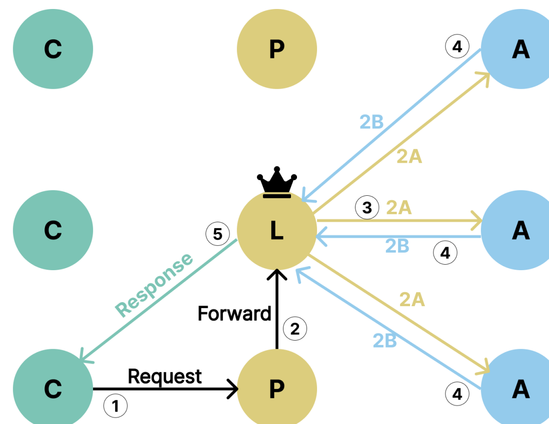
Quanto ao número de mensagens, em sistemas de alta vazão, devido a limitações de *hardware*, como taxa de transferência da rede, espaço de *buffers* e tempo de processamento, um único nodo não se mostra capaz de processar todas as mensagens necessárias para se alcançar o consenso (MAO; JUNQUEIRA; MARZULLO, 2008; AILIJIANG; CHARAPKO; DEMIRBAS; KOSAR, 2020; MORARU; ANDERSEN; KAMINSKY, 2013; MARANDI; PRIMI; PEDONE, 2012), de forma que estratégias que reduzam o número de mensagens

utilizadas ou distribua o custo de processamento entre os nodos são utilizadas (MARANDI; PRIMI; SCHIPER *et al.*, 2010; CHARAPKO; AILIJIANG; DEMIRBAS, 2021).

3.2.1 *Fast* Paxos

O algoritmo Multi-Paxos tem como otimização a reserva de múltiplos números de proposta futuros por meio de uma única fase 1 do algoritmo Paxos. Isto permite que decisões, após a escolha de um líder, sejam tomadas em apenas duas rodadas de mensagens (`2A` e `2B`), o número mínimo de rodadas possível de acordo com Lamport (2000). No entanto, em situações práticas, uma rodada adicional de mensagens pode ser necessária para que um *proposer* P, após receber um comando de um cliente C, encaminhe este para que o líder L atual proponha o comando para os *acceptors* A. Isto pode ser visto na Figura 5, onde, após o recebimento de um comando a ser proposto (`Request`), ao invés de propo-lo diretamente para um *aceptor*, o *proposer* P deve encaminhar (`Forward`) este comando para o líder do protocolo, que então propõe ele e, após este ser decidido, responde o cliente.

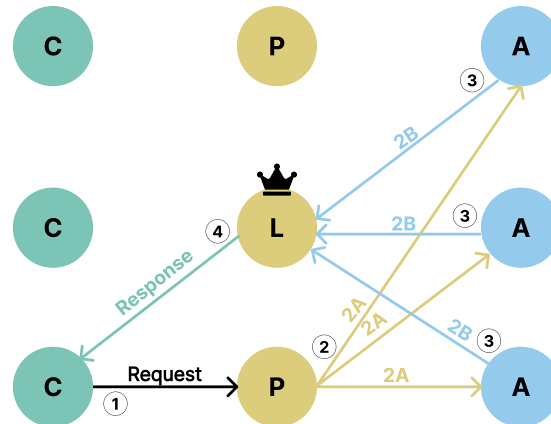
Figura 5 – Rodadas de comunicação no Multi-Paxos (assumindo que um líder existe)



Fonte: De autoria própria.

O *Fast* Paxos (LAMPORT, 2006) busca remover esta rodada adicional de mensagens ao permitir que qualquer *proposer* envie suas propostas diretamente para um quorum de *acceptors*. Para isso, esta variante divide os números de proposta em números de proposta rápidos e clássicos. Em uma rodada rápida, o líder pode, ao invés de propor um valor, enviar uma mensagem `2A` com o valor "qualquer". Esta mensagem informa aos *acceptors* que estes podem aceitar o primeiro valor que receberem de outro *proposer* como se fosse enviado pelo líder. Já rodadas clássicas funcionam igual ao algoritmo Paxos original. Como pode ser visto na Figura 6, agora, ao receber um comando de um cliente, o *Proposer* pode imediatamente enviar uma mensagem `2A` aos *acceptors*, que então encaminham suas respostas `2B` ao líder para este determinar se o comando foi decidido ou não.

Figura 6 – Rodadas de comunicação em uma rodada rápida



Fonte: De autoria própria.

A introdução de rodadas rápidas traz como potencial problema o caso em que múltiplos valores sejam propostos, por diferentes *proposers*, ao mesmo tempo e que nenhum valor seja aceito por uma maioria de *acceptors*. No caso desta ocorrência, chamada de colisão, o líder, com base nas mensagens 2B que recebeu dos *acceptors*, seleciona um valor seguro a ser proposto e o propõe em uma nova rodada clássica. Para que o líder possa selecionar um valor seguro, ou seja, que não leve à inconsistências, é necessário alterar as regras de seleção de quorums.

Assim, no *Fast* Paxos, além de necessitar que dois quorums quaisquer possuam uma intersecção entre si, dois quorums j de rodadas rápidas quaisquer também devem possuir uma interseção com qualquer outro quorum i. A partir desta restrição, de forma a tolerar o maior número de falhas com N nodos, quorums clássicos possuem tamanho $(N/2) + 1$ e quorums rápidos possuem tamanho $3N/4$, ou seja, para que o líder considere um valor como decidido em uma rodada rápida, este deve receber confirmação de no mínimo três quartos dos *acceptors*. Também é possível reduzir o tamanho de quorums rápidos, tal que ambos os tipos de quorums possuam tamanho $(2N/3) + 1$, mas reduzindo a quantidade de falhas toleradas.

3.2.2 *Flexible* Paxos

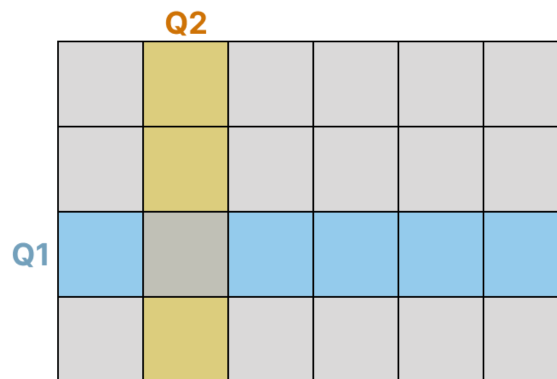
No algoritmo Paxos original, uma maioria de nodos é necessária para formar quorums para as fases 1 e 2 do protocolo, de forma que, em um sistema com N nodos, no mínimo $(N/2) + 1$ mensagens devem ser enviadas por um *proposer* em cada fase. No entanto, conforme observado em Lamport, Malkhi e Zhou (2009b), esta composição é mais conservadora do que o necessário para se garantir as propriedades de um protocolo de consenso regular. Mais especificamente, o *Flexible* Paxos (LAMPORT; MALKHI; ZHOU,

2009b) conclui que, para garantir uma execução consistente, a única característica necessária é que todos os quorums da fase 1 do protocolo possuam uma intersecção com todos os quorums da fase 2.

A partir desta definição de quorums, denominados de quorums flexíveis, é possível, por exemplo, reduzir o número de nodos necessários para um quorum da fase 2 (Q2), ao aumentar o número de nodos em quorums da fase 1 (Q1). Como a fase 2 do algoritmo é executada mais frequentemente do que a fase 1, que ocorre apenas em casos de falha do líder, esta alteração reduz o número de mensagens trocadas e aumenta a vazão do protocolo. No entanto, o algoritmo tolera um número menor de falhas, já que mais nodos ativos são necessários para formar um quorum da fase 1.

Também é possível criar quorums com diferentes estruturas e propriedades, como é o caso de quorums de grade (*grid quorums*), que reduzem o tamanho de ambos Q1 e Q2. Este modelo de quorums organiza todos os nodos em uma grade, e define as linhas desta como possíveis quorums da fase 1 e as colunas como possíveis quorums da fase 2, garantindo que todos os quorums Q1 possuam intersecção com todos os quorums Q2, conforme visto na Figura 7.

Figura 7 – Exemplo de uma configuração de quorums de grade 4x5



Fonte: Adaptado de Lamport, Malkhi e Zhou (2009b).

Ao criar quorums a partir desta configuração em grade, também deixamos de tratar todas as falhas no sistema como iguais, de forma que, mesmo com o mesmo número de falha ocorrendo, dependendo da configuração dos nodos, esta pode ou não impedir o progresso do protocolo. Na Figura 8, por exemplo, é possível ver que apesar de 5 nodos terem falhados nos quatro casos apresentados, dependendo dos nodos específicos que falharam diferentes situações podem surgir:

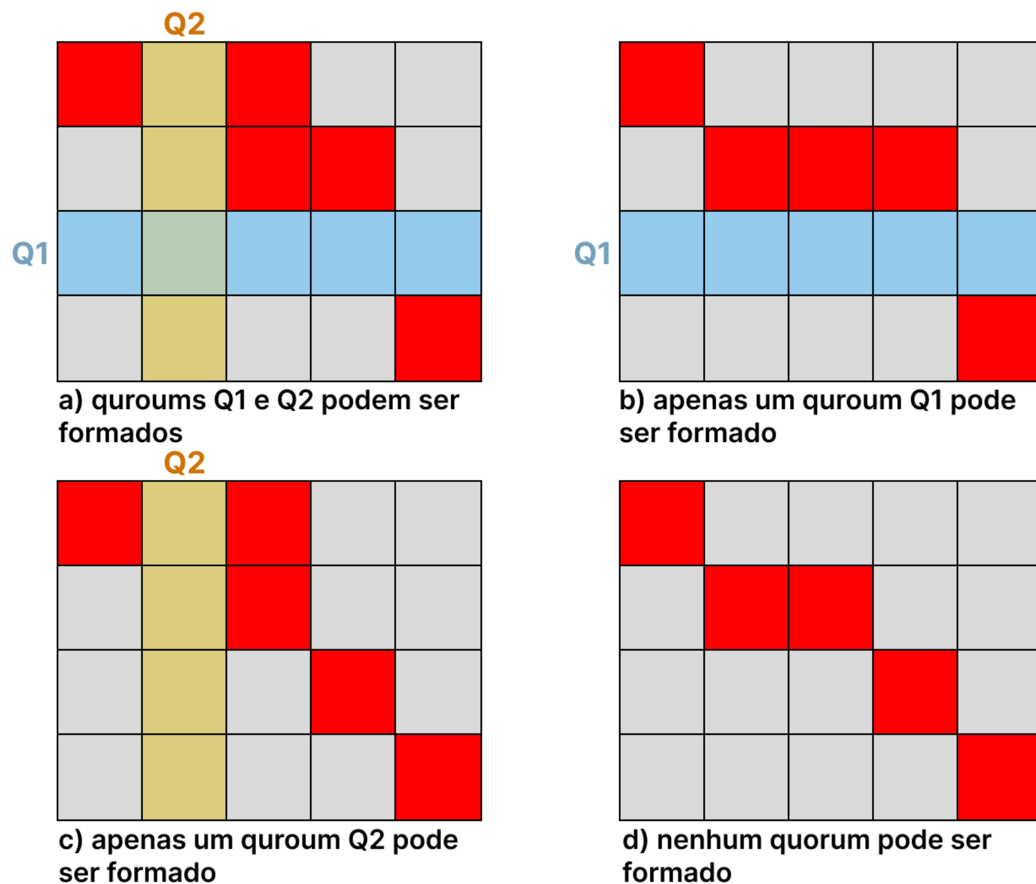
a) A execução do algoritmo segue normal, pois ainda é possível formar quorums Q1 (já que existe uma linha sem nodos defeituosos) e Q2 (pois existe uma coluna sem nodos defeituosos).

b) O algoritmo não pode mais progredir, pois não é possível formar um quorum Q2.

c) O algoritmo ainda pode decidir comandos (executar a fase 2), desde que não tenha ocorrido a falha de um líder e necessidade de uma eleição (executar a fase 1).

d) O algoritmo não pode mais progredir, pois não é possível formar um quorum Q1 ou Q2.

Figura 8 – Tipos de falhas possíveis em quorums de grade



Fonte: De autoria própria.

3.2.3 *Ring* Paxos

Os líderes no Paxos podem facilmente se tornar o gargalo do algoritmo, ao não conseguir processar, receber ou enviar todas as mensagens necessárias. Tendo em vista este problema, o *Ring* Paxos (MARANDI; PRIMI; SCHIPER *et al.*, 2010) é um variante do protocolo Paxos que busca melhor distribuir a carga de processamento de mensagens entre os nodos, evitando que o líder se torne um gargalo. Para isso, este algoritmo modifica a forma como as mensagens são enviadas, por meio de duas alterações principais: o envio

de mensagens 2B em um anel lógico e o envio de mensagem 2A e 3 via *multicast* a nível de rede (*ip-multicast*).

Nesta variante, durante a fase 1 do protocolo, quando um líder é selecionado este também propõe uma estrutura de anel lógico para o envio de mensagens. Este anel tem como forma uma sequência de nodos que formem um quorum de *acceptors* e que termine no nodo líder. Este anel é então utilizado durante a fase 2 do algoritmo, conforme visto na Figura 9, na qual:

1. O **líder** envia uma mensagem 2A para todos os *acceptors* que participam do anel.
2. O **primeiro *acceptor*** (A1) da sequência, após receber uma mensagem 2A do líder, envia sua mensagem 2B para o próximo nodo da sequência.
3. Um ***acceptor*** (A2), ao receber a mensagem 2B do nodo antecedente, envia sua mensagem 2B para o próximo nodo da sequência.
4. Um ***acceptor*** (A3), ao receber a mensagem 2B do nodo antecedente, envia sua mensagem 2B para o último nodo da sequência, o líder.
5. O **líder**, ao receber uma mensagem 2B de seu antecedente, sabe que o comando foi decidido.

Esta mudança na fase 2 faz com que o líder deixe de ser um gargalo para o recebimento de mensagens, já que agora, ao invés de receber, para cada mensagem 2A, uma resposta de cada *acceptor*, este apenas recebe uma resposta do penúltimo nodo do anel lógico e assume que os nodos antecedentes também aceitaram o valor proposto. Caso um nodo falhe e o anel lógico não possa mais ser completado, o líder ou outro *proposer* executa novamente a fase 1 do protocolo, propondo um novo anel lógico.

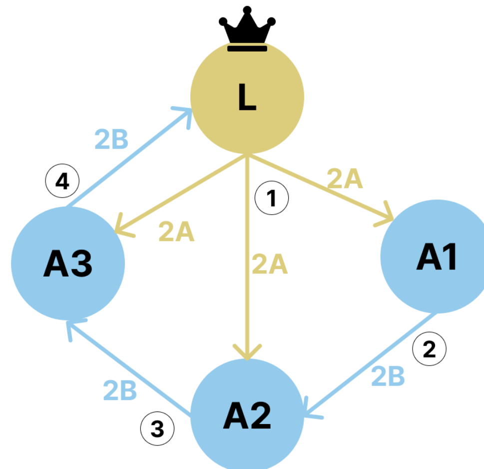
Além disso, o custo de envio das mensagens 2A e 3 (*commit*) também é reduzido por meio do uso de *multicast*, com o qual o líder é capaz de se comunicar com todos os *acceptors* e *learners* em uma única mensagem, conforme pode ser visto nos passos 2 e 5 do exemplo c da Figura 10. Além disso, o uso de *multicast*, quando realizado por um único processo por vez, também maximiza a vazão de mensagens e possui melhor escalabilidade em relação ao número de nodos do que o envio via *unicast*(MARANDI; PRIMI; SCHIPER *et al.*, 2010).

3.2.4 *Pig* Paxos

Similar ao *Ring* Paxos, o *Pig* Paxos também altera a forma como mensagens são enviadas de forma a distribuir mais igualmente a carga do processamento de mensagens, assim evitando que o líder se torne um gargalo.

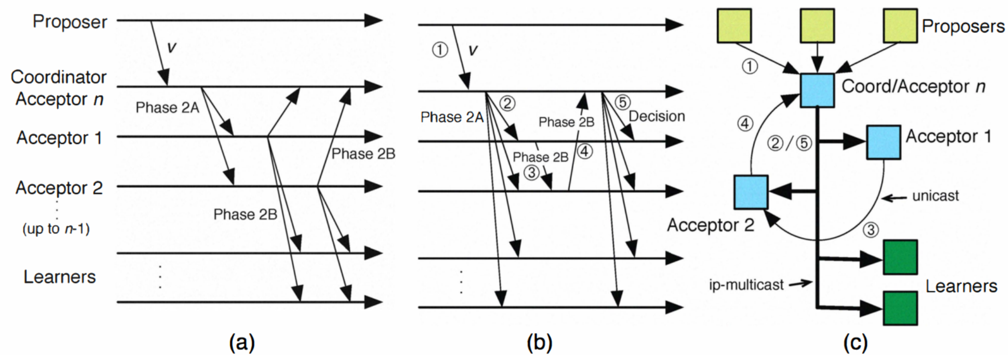
Para alcançar este objetivo, esta variante organiza todos os nodos em distintos grupos de redistribuição, seja por meio de uma função *hash* ou em grupos pré-definidos.

Figura 9 – Mensagens da fase 2 do *Ring* Paxos (Com anel lógica A1 → A2 → A3 → L)



Fonte: De autoria própria.

Figura 10 – Paxos clássico (a) e *Ring* Paxos (b,c)

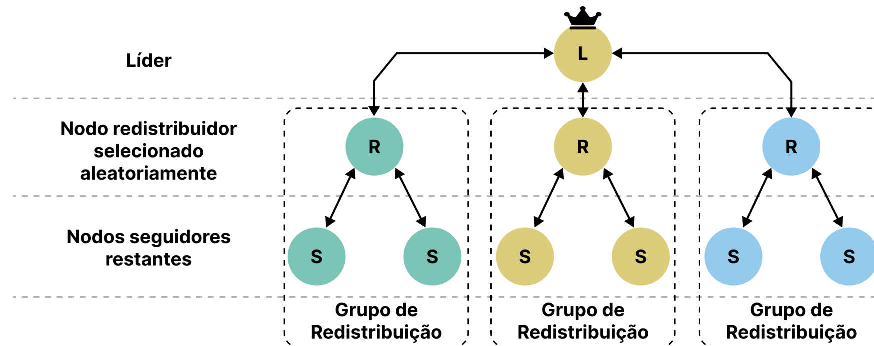


Fonte: (MARANDI; PRIMI; SCHIPER *et al.*, 2010)

Como pode ser visto na Figura 11, quando um nodo L precisa enviar uma mensagem 1A, 2A ou 3, ele aleatoriamente seleciona, para cada grupo de redistribuição, um integrante que irá agir como redistribuidor R. O **redistribuidor** tem três funções no *Pig* Paxos:

- Processar normalmente a mensagem recebida.
- Reenviar a mensagem recebida para os outros integrantes do grupo F.
- Agrupar as respostas recebidas e enviá-las, como uma única mensagem, para o nodo que iniciou a comunicação.

Figura 11 – Envio de mensagens no *Pig* Paxos



Fonte: Adaptado de Charapko, Ailijiang e Demirbas (2021)

Apesar de reduzir a quantidade de mensagens que um único nodo precisa processar, evitando assim o gargalo no líder, o *Pig* Paxos também aumenta a quantidade de rodadas de comunicação necessárias entre os nodos e, com seu modelo de redistribuição, introduz novos tipos de falhas que devem ser consideradas. Dentre estas falhas, temos grupos de redistribuição incompletos, que então não respondem ao líder ou devem mandar uma resposta incompleta após um período de tempo t, e a falha de um nodo selecionado como redistribuidor, que deve ser detectada para que um novo redistribuidor possa ser selecionado para aquele grupo.

3.3 ALGORITMOS MULTI-LÍDER

Conforme apontado por Mao, Junqueira e Marzullo (2008) e Moraru, Andersen e Kaminsky (2013), outro problema encontrado em protocolos de consenso trata-se da escalabilidade de sua vazão em proporção com o número de réplicas. Em protocolos de consenso como o Paxos, a adição de novas réplicas, ao invés de aumentar a vazão do algoritmo, acaba por prejudicá-la, devido ao fato de aumentar o número de mensagens que o líder precisa enviar e processar, sem aumentar a vazão do algoritmo.

Neste contexto, algoritmos de consenso multi-lider são aqueles que buscam aumentar sua vazão em proporção com o número de réplicas presentes no sistema, ao permitir que múltiplos nodos atuem simultaneamente como líder do protocolo de consenso, propondo e decidindo novos comandos. Para isso, estratégias diversas podem ser utilizadas, como dividir o espaço do *log* (MAO; JUNQUEIRA; MARZULLO, 2008; STATHAKOPOULOU; PAVLOVIC; VUKOLIĆ, 2022) ou explorar a semântica da aplicação sendo replicada (MORARU; ANDERSEN; KAMINSKY, 2013; AILIJIANG; CHARAPKO; DEMIRBAS; KOSAR, 2020; MARANDI; PRIMI; PEDONE, 2012).

Desta forma, nesta seção apresentamos alguns trabalhos que exploram técnicas voltadas para o aumento da escalabilidade e vazão de protocolos de consenso distribuídos

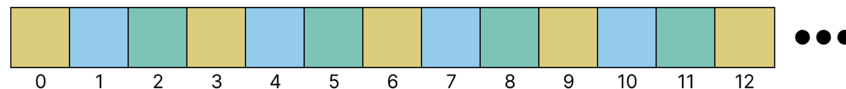
por meio da introdução de múltiplos líderes.

3.3.1 Mencius

O protocolo Mencius (MAO; JUNQUEIRA; MARZULLO, 2008), utiliza uma das abordagens mais intuitivas para se permitir múltiplos líderes a partir do algoritmo multi-paxos, particionar o espaço do *log*. Nesta variante, cada nodo participando do protocolo é líder de uma partição do espaço de *log* específica. Esta estratégia, além de limitar o gargalo no processo líder, também reduz a latência do protocolo ao permitir a todos os processos diretamente propor seus valores aos *acceptors*, sem necessidade de encaminhá-los ao líder.

No Mencius, as partições do *log* são atribuídas aos nodos de maneira prévia, de forma que todos os nodos conhecem o mapeamento. Partições são atribuídas por rodízio, tal que, em um sistema com 3 réplicas, o nodo 0 é responsável pelas posições 0, 3, 6, 9, ... do *log*, o nodo 1 é responsável pelas posições 1, 4, 7, 10, ... e o nodo 2 é responsável pelas posições 2, 5, 8, 11, ..., conforme ilustrado na Figura 12.

Figura 12 – Partição do *log* no Mencius (cores indicam atribuição à um nodo)



Fonte: De autoria própria.

No entanto, particionar o *log* desta forma introduz um novo problema. Como os líderes propõem comandos em velocidades diferentes, lacunas são formadas no *log* e, como os comandos precisam ser executados sequencialmente, mesmo que um comando n já tenha sido decidido pelo líder de sua partição, este deve esperar até que os comandos 0 até $n-1$ sejam decididos pelos outro líderes para finalmente poder executar o comando n. Para evitar esta situação, onde o líder mais lento determina a vazão máxima do algoritmo, o Mencius introduz os comandos *no-op*.

O *no-op* é um comando que pode ser adicionado no *log* para indicar que nenhuma mudança precisa ser feita na máquina de estados. Este comando é proposto por um líder quando este percebe que está atrasado em relação ao líder de outra partição. Por exemplo, caso o líder L tenha proposto apenas até a posição i do *log* e receba uma mensagem 2A de outra réplica referente a uma posição j maior do i, o líder L então envia mensagens 2A propondo um valor *no-op* para todas as posições do *log* menores que j para as quais ainda não tenha proposto.

Outra alteração em relação ao protocolo Paxos original está na forma como o Mencius lida com a falha de um dos seus líderes. Um nodo P, ao suspeitar outro nodo de ter falhado, executa a fase 1 do algoritmo Paxos normalmente, no entanto, caso seja

selecionado como líder, P apenas pode propor um valor previamente proposto pelo líder daquela partição ou, caso nenhum valor tenha sido previamente proposto, um valor *no-op*. Com esta restrição no que pode ser proposto, assim que um valor *no-op* é enviado em uma mensagem 2A, o líder já pode considerá-lo decidido e outro nodo, ao receber a mensagem 2A, também pode imediatamente aprender o valor, de forma que não são necessárias mensagens 2B ou 3 para que estes valores sejam decididos.

É neste método de lidar com a falha de líderes que se encontra um dos principais problemas do protocolo Mencius proposto em Mao, Junqueira e Marzullo (2008). Diferentemente da versão padrão de otimização do multi-paxos, onde a fase 1 torna um determinado processo líder do algoritmo para todas as instâncias subsequentes, no Mencius a fase 1 deve ser executada para cada instância ou, em versões otimizadas, a fase 1 é realizada de forma que o processo se torne líder por n instâncias. No entanto, caso o líder original daquela partição não se recupere ou leve um tempo longo para se recuperar, os outros nodos devem ficar executando a fase 1 do algoritmo indefinidamente e propondo valores *no-op* na partição inativa, fazendo com que qualquer falha se mostre devastadora para o protocolo como um todo e levando o protocolo a não ser tão utilizado na prática (CHARAPKO, 2024).

3.3.2 *Multi-Ring* Paxos

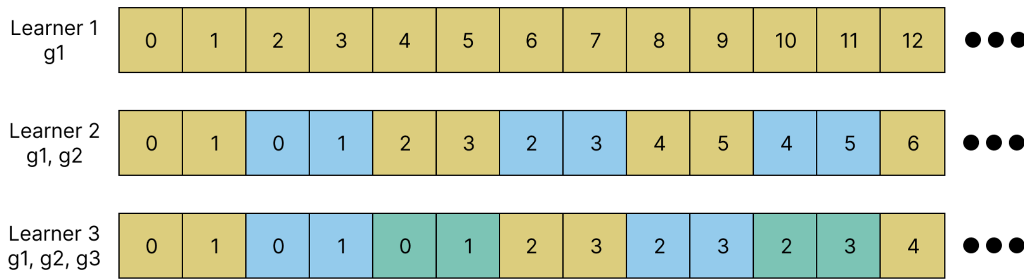
O *Multi-Ring* Paxos (MARANDI; PRIMI; PEDONE, 2012) também é uma variante que busca melhorar a escalabilidade do protocolo Paxos. Para isso, ele executa múltiplas instâncias paralelas do *Ring* Paxos e ordena os resultados destas instâncias por meio de um algoritmo determinístico.

Cada instância do *Ring* Paxos é responsável por ordenar os comandos de um determinado grupo. Um grupo é uma partição do espaço de comandos da aplicação. Por exemplo, em um banco de dados, comandos podem ser particionados de acordo com o valor da chave que este comando altera. Com isso, cada *learner* no *Multi-Ring* Paxos pode participar de um ou mais grupos, tal que replique apenas determinadas partições da aplicação.

De forma a unificar a sequência de comandos decididos por diferentes grupo e garantir que exista uma ordem relativa entre os comandos, cada *learner* executa um algoritmo de rodízio, onde entrega uma certa quantidade m de comandos decididos por cada um de seus grupos em uma ordem pré-definida. Por exemplo, na figura 13, o *learner* 1 participa de um único grupo (g1), então apenas entregas os comandos dele. Já os *learners* 2 e 3, que participam de mais de um grupo, entregam 2 mensagens de cada grupo em determinada ordem (g1 $\rightarrow$ g2 $\rightarrow$ g3), de forma que a mensagem 0 de g1 sempre seja entregue antes da mensagem 0 de g2, garantindo que a ordem relativa dos comandos seja igual para todos os nodos.

Assim como o Mencius, grupos diferentes podem decidir comandos em velocidades

Figura 13 – Ordenação de mensagens do *Multi-Ring* Paxos (m=2)

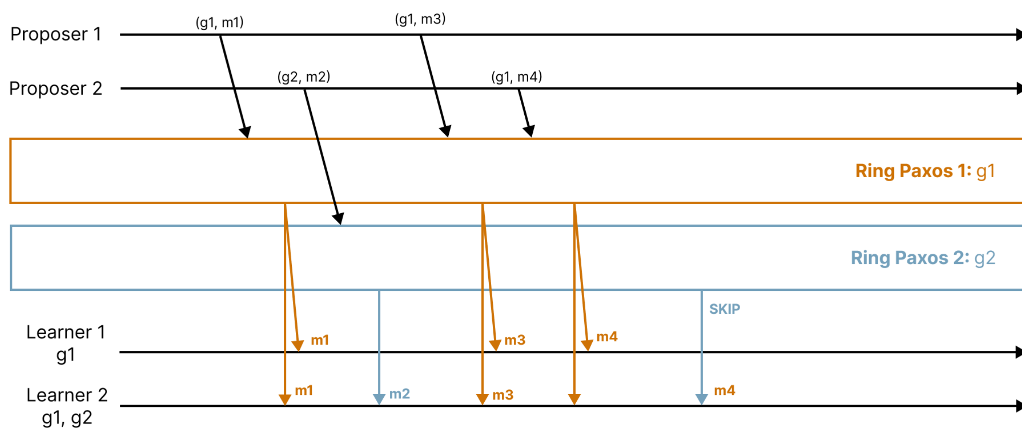


Fonte: De autoria própria.

desiguais, formando lacunas de comandos em *learners* que estejam participando de ambos. Para resolver este problema, um parâmetro λ que representa a vazão máxima esperada de um grupo qualquer é definido previamente. Periodicamente, o líder de cada grupo compara λ à sua própria vazão σ, sendo que, caso $\lambda > \sigma$, ele propõe valores *no-op* (mensagem *SKIP*) o suficiente para alcançar a vazão máxima esperada.

Como exemplo do mecanismo utilizado para evitar lacunas, na figura 14, o grupo g1 decide 3 valores enquanto g2 decide apenas 1, impedindo que o *learner* 2 aprenda o valor da mensagem $m4$ até que receba um segundo valor do grupo 2. O coordenador de g2, após algum tempo, percebe que possui vazão abaixo do esperado e envia uma mensagem *SKIP*, possibilitando ao *learner* 2 finalmente aprender aprender $m4$.

Figura 14 – Aprendendo valores no *Multi-Ring* Paxos



Fonte: Adaptado de Marandi, Primi e Pedone (2012)

Por fim, uma otimização comum no *Multi-Ring* Paxos é a utilização de *batches*.

Batches agrupam até n comandos em uma única estrutura, que pode então ser proposta em uma única instância do protocolo de consenso, reduzindo a quantidade de mensagens necessária para cada decisão. Ao receber um novo comando ao ser proposto, os líderes, ao invés de imediatamente enviá-lo, apenas adicionam este ao *batch* e esperam até que este esteja cheio ou que uma quantidade de tempo δ tenha se passado para enfim propô-lo. Esta proposta tem ótima compatibilidade com o *Multi-Ring* Paxos pois, caso nenhum comando tenha sido recebido no intervalo de tempo δ, o líder pode propor um *batch* vazio, que atua como uma mensagem *SKIP* e garante uma vazão mínima ao algoritmo.

3.3.3 WAN Paxos (WPaxos)

O WAN Paxos (WPaxos) (AILIJIANG; CHARAPKO; DEMIRBAS; KOSAR, 2020) é uma variante do protocolo Paxos que busca resolver o problema da baixa vazão presente em algoritmos de consenso quando implantados em *Wide Area Networks* (WAN) e, de forma mais específica, prioriza aplicações para bancos de dados chave-valor.

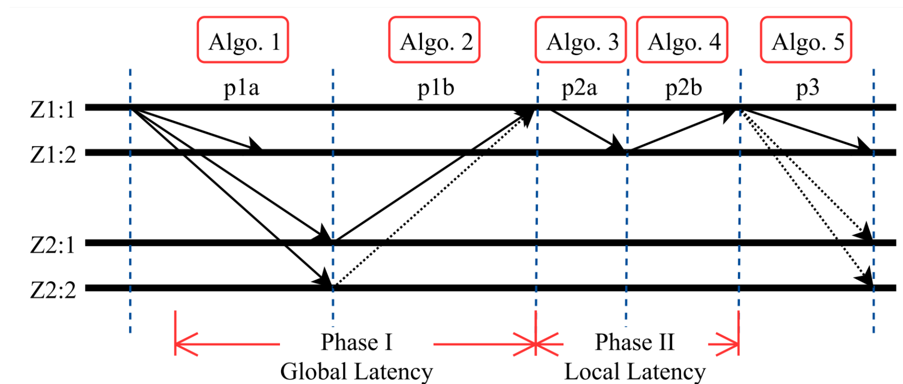
Seu principal método para aumento da escalabilidade envolve a criação de um *log* independente para cada objeto presente no banco de dados chave-valor. Assim, uma instância do protocolo Paxos é executada para cada um dos objetos e cada nodo de execução do WPaxos torna-se o líder de um subconjunto específico destes objetos.

Caso um processo receba um pedido relacionado a um objeto do qual não é líder, ele pode então encaminhar a requisição para o atual líder do objeto ou utilizar uma mensagem 1A do protocolo Paxos para se tornar o líder da instância relacionada àquele objeto e então diretamente propor um novo comando. De forma a evitar que a fase 1 seja executada muito frequentemente, ela geralmente é iniciada pelo líder atual do objeto ao perceber que outro nodo se beneficiaria mais como líder do objeto, com outros nodos apenas executando-a espontaneamente quando suspeitam da falha do líder atual.

Além disso, o WPaxos também apresenta otimizações voltadas para sistemas geograficamente distribuídos. Para isso, ele se utiliza do conceito de quóruns flexíveis, apresentados em Lamport, Malkhi e Zhou (2009b), para formar quorums Q2 geograficamente próximos do líder, evitando assim grandes latências e possibilitando o aumento da vazão. Como pode ser visto na figura 15, durante a fase 2 do algoritmo, mensagens são enviadas apenas para nodos na mesma zona geográfica Z1, resultando em menores latência. Apenas no caso de eleição de um novo líder para um objeto ou de falha de nodos mais próximos que processos do WPaxos precisam utilizar Q1 e enviar mensagens para nodos geograficamente distantes.

Durante os experimentos de Ailijiang, Charapko, Demirbas e Kosar (2020), foi encontrado que, em situações onde alguns objetos são acessados mais frequentemente em locais próximos, o WPaxos consegue oferecer uma melhor vazão e menor latência média que outros algoritmos utilizados para implementar o consenso de sistemas geograficamente distribuídos, como o *Egalitarian* Paxos (MORARU; ANDERSEN; KAMINSKY, 2013).

Figura 15 – Fluxo de mensagens no WPaxos



Fonte: Ailijiang, Charapko, Demirbas e Kosar (2020)

Apesar de trazer resultados promissores, a estratégia utilizada pelo WPaxos possui algumas limitações inerentes. Uma delas é a necessidade do protocolo de identificar qual objeto será afetado por um determinado comando. A outra limitação, derivada do fato do *log* de cada objeto ser independente, é que os comandos da aplicação só podem afetar um único objeto por vez, limitando assim as aplicações nas quais pode ser utilizado.

Para os fins deste trabalho, o WPaxos será utilizado para a comparação de desempenho entre este e o protocolo implementado. Ele foi escolhido por se tratar de um protocolo multi-líder de alto desempenho que apresenta uma proposta similar de paralelização àquela explorada neste trabalho ao possuir um líder para cada subdivisão do *log*.

3.3.4 Egalitarian Paxos (EPaxos)

O Egalitarian Paxos (EPaxos), proposto por Moraru, Andersen e Kaminsky (2013), implementa uma variante do protocolo Paxos que procura distribuir igualmente a carga de trabalho para todas as suas réplicas. Com isto, busca-se aumentar a escalabilidade e vazão do protocolo, assim como tolerar réplicas lentas ou falhas em algumas réplicas com uma degradação de performance tolerável.

Similarmente ao *Fast* Paxos, todos os nodos do EPaxos podem encaminhar suas propostas diretamente para um *acceptor*. O principal ponto de diferenciação do EPaxos em relação aos outros algoritmos de consenso apresentados, é o fato deste não tomar decisões sobre a ordem absoluta na qual os comandos serão executados. Ao invés disso, para cada comando recebido, o EPaxos decide quanto as dependências deste comando, ou seja, que comandos já foram aceitos que entram em conflito com o comando atual e devem ser executados antes desse, de forma a criar uma ordem parcial de execução que não leve a violação de consistência. Além disso, também é decidido um número de sequência para

o comando, utilizado para quebrar ciclos no grafo de dependência.

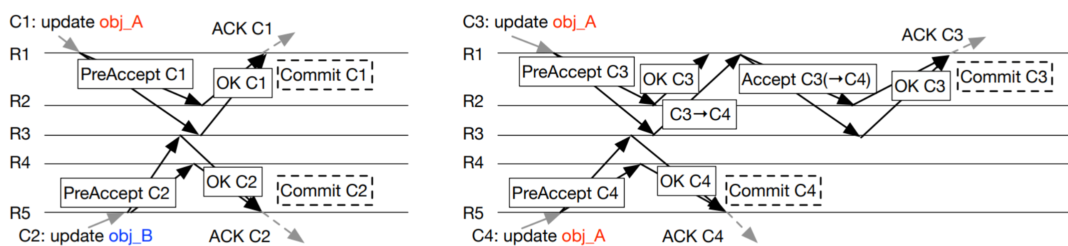
Ao receber um comando c, um *proposer* calcula suas dependências $deps_c$ (todos os comandos previamente conhecidos que alteram a mesma variável) e um número de sequência $seqs_c$ $(1 + max(seq_d \cup \{0\}), d \in deps_c)$ e os envia em uma mensagem *Pre-accept* para os *acceptors*. Um *acceptor*, ao receber a mensagem *Pre-accept* atualiza os valores *deps* e *seqs* desta, levando em consideração os valores que este conhece, e envia estes valores como resposta. Ao receber um quorum rápido, similar ao do *Fast Paxos*, de pelo menos $3N/4$ nodos, o *proposer* então decide se este comando pode ser decidido no *Fast Path* ou se deve seguir no *Slow Path*.

O *Fast Path* é utilizado quando todas as respostas dos *acceptors* são iguais, ou seja, todos concordam quanto às dependências e ao número de sequência do comando c. Neste caso, o comando pode entrar para a fila de execução da máquina, tendo sido definitivamente decidido, e o *proposer* envia mensagens de *commit* (P3) para os outros nodos.

Caso os participantes não consigam concordar imediatamente quanto as dependências do comando devido ao fato de outro comando, que interfere com este, ter sido proposto e ser conhecida por uma parte do quorum, o comando deve seguir no *Slow Path*. Neste caso, uma fase 2 padrão do Paxos é executada, com o *proposer* propondo a união de todas as dependências e o maior número de sequência presente nas respostas, com as dependências do comando sendo decididas apenas após uma maioria de respostas.

Na Figura 16, temos, no exemplo à esquerda, dois comandos não conflitantes, C1 e C2, sendo ambos decididos em um *Fast Path*, já que operam em objetos diferentes e não possuem dependências entre si. Já no exemplo à direita, temos os comandos conflitantes C3 e C4. O comando C4 é decidido em um *Fast Path* pela réplica R5, no entanto, o comando C3 precisa ser decidido por meio de um *Slow Path*, pois a dependência C3 $\rightarrow$ C4 foi recebida como resposta.

Figura 16 – Fluxo de mensagens no EPaxos



Fonte: Moraru, Andersen e Kaminsky (2013)

Devido às características descritas, o desempenho do EPaxos depende diretamente da distribuição dos comandos da aplicação. Caso uma grande porção dos comandos entre em conflito uns com os outros, mais rodadas do protocolo de consenso serão necessárias

para decidir as dependências do comando, reduzindo a vazão do algoritmo e levando a uma situação similar ao *livelock*. Apesar disso, em situações onde 2% dos comandos conflitam, o EPaxos ainda possui melhor performance que algoritmos utilizados comercialmente, como o Mencius (MAO; JUNQUEIRA; MARZULLO, 2008) e multi-Paxos (LAMPORT, 2001), devido ao fato de processar menos mensagens por comando que o primeiro e por não possuir um único líder como o segundo.

O EPaxos será utilizado durante os experimentos como forma de comparação entre o desempenho deste em relação ao do algoritmo implementado. O EPaxos foi selecionado por apresentar uma proposta de paralelização diferente daquela explorada neste trabalho ao explorar a semântica da aplicação e se tratar de uma abordagem sem líderes.

4 IMPLEMENTAÇÃO DO PAXOS MULTI-LÍDER

Conforme destacado no Capítulo 3, algoritmos de consenso que possuem um único líder apresentam uma clara degradação de desempenho com o aumento do número de nodos, já que a sua vazão não escala proporcionalmente à adição de novos nodos devido a um gargalo no líder. Uma das possíveis soluções para este problema é a utilização de algoritmos multi-líder, que melhor distribuem o custo de execução do protocolo de consenso entre os nodos (MORARU; ANDERSEN; KAMINSKY, 2013; AILIJIANG; CHARAPKO; DEMIRBAS; KOSAR, 2020; MARANDI; PRIMI; PEDONE, 2012).

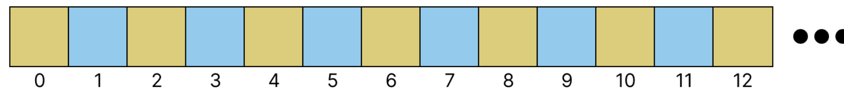
Neste contexto, o *Insanely Scalable SMR* (ISS), proposto em Stathakopoulou, Pavlovic e Vukolić (2022), é um *framework* capaz de paralelizar a execução de algoritmos de consenso com um único líder, permitindo que estes se comportem como algoritmos multi-líder. Assim, a aplicação do ISS pode aumentar a escalabilidade destes algoritmos, ao melhor distribuir a carga de processamento de mensagens e reduzir o gargalo num único processo líder.

O princípio por trás do ISS está em arquitetar a execução paralela de múltiplas instâncias de um protocolo de consenso qualquer. Cada uma das instâncias do algoritmo de consenso implementa o que é chamado de *Sequenced Broadcast* (SB), onde o protocolo é inicializado com um líder específico e, após este decidir n valores, a instância é terminada. No SB, assim como no Mencius (MAO; JUNQUEIRA; MARZULLO, 2008), apenas este líder inicial pode propor comandos, com outros nodos que assumam a posição por meio de uma eleição devendo propor valores vazios. Para garantir que esses comandos decididos por instâncias paralelas sejam executadas na mesma ordem por todos os nodos participando do algoritmo de consenso, o ISS utiliza um *log* particionado em segmentos.

O *log* é um espaço contínuo de comandos de clientes, onde cada posição é identificada por um número de sequência, atribuído de forma crescente e ordenada a partir do 0, e podendo conter um comando a ser executado. Já segmentos são partições sem sobreposição deste *log*, criadas, assim como no Mencius (MAO; JUNQUEIRA; MARZULLO, 2008), ao atribuir os números de sequências do *log* a segmentos por meio de um simples carrossel. Por exemplo, considerando a existência de dois segmentos, podemos dividir o *log* tal que o segmento 0 é composto pelas posições 0, 2, 4, $\ldots$ do *log* e o segmento 1 é composto pelas posições 1, 3, 5,..., conforme pode ser visto na Figura 17. Com os segmentos estabelecidos, associamos cada instância do algoritmo de consenso a um único segmento. Esta instância fica então responsável por decidir todos os comandos para as posições do *log* que compõem seu segmento. Assim, decisões de uma instância nunca interferem nas decisões de outra instância e uma ordem absoluta de execução é estabelecida por meio dos números de sequência.

No entanto, esta estratégia traz novos problemas de consistência que devem ser tratados. Em primeiro lugar, como as instâncias são isoladas, caso os comandos de um

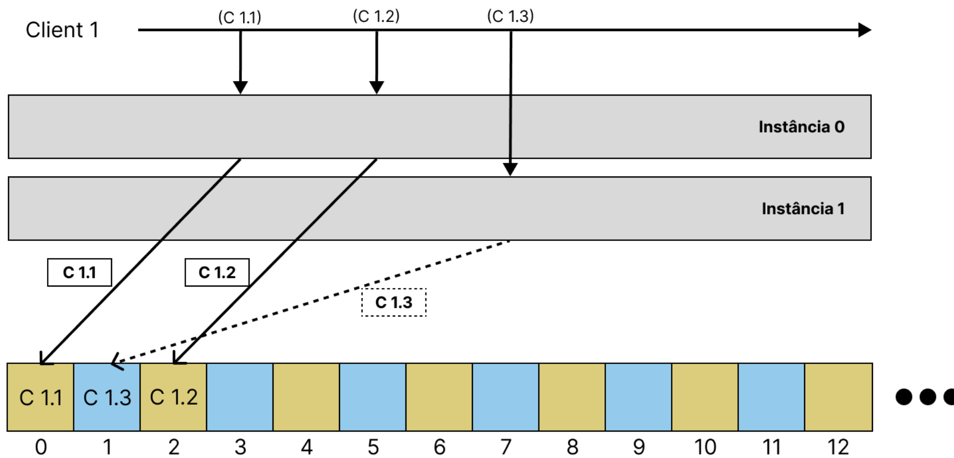
Figura 17 – Divisão do *log* em segmentos



Fonte: De autoria própria.

cliente sejam mandados para segmentos diferentes, estes podem ser decididos fora de ordem, levando a inconsistências indesejadas. Por exemplo, na Figura 18, temos uma situação onde o Cliente 1, após ter seus primeiros comandos (C 1.1 e C1.2) decididos nas posições 0 e 2 do *log* pela instância responsável pelo segmento 0, envia seu terceiro comando (C 1.3) para o segmento 1 do algoritmo de consenso, que então decide colocá-lo na posição 1 do *log*. Nesta situação, o comando 1.3 é executado antes do 1.2, uma situação que pode levar a inconsistências durante a replicação da aplicação.

Figura 18 – Possível erro de ordenação

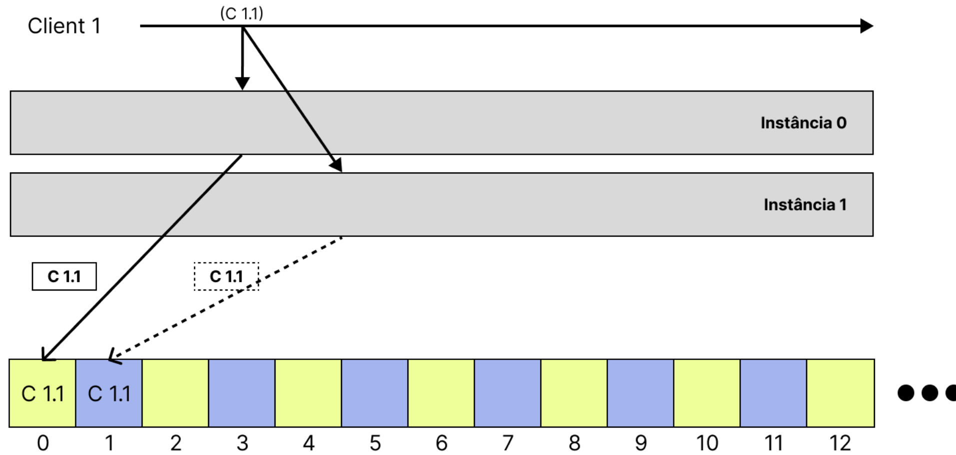


Fonte: De autoria própria.

Outro potencial problema é a duplicação de comandos, onde segmentos distintos decidem o mesmo comando *c* para duas ou mais posições do *log*. Conforme visto na Figura 19, caso C1 envie o seu comando C1 para ambas as instâncias 0 e 1, este será decidido em duas posições distintas do *log*, já que as instâncias não se comunicam entre si. Para evitar problemas de ordenação e duplicação de comandos, o ISS introduz o conceito de *buckets*.

Um *bucket* é uma estrutura de dados para onde pedidos de um subconjunto de clientes são encaminhados. Todos os pedidos de um determinado cliente devem ser direcionados para um mesmo *bucket*, conforme pode ser visto na Figura 20. Nesta figura, os pedidos

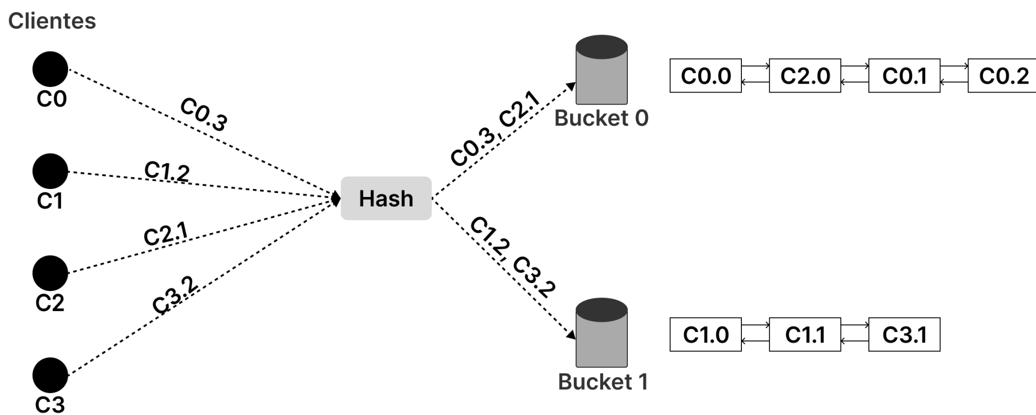
Figura 19 – Possível erro de duplicação



Fonte: De autoria própria.

são enviados para seus *buckets* correspondentes por meio de uma função de *hash*, com pedidos de clientes com identificadores par sendo enviados para o *Bucket* 0 e aqueles com identificadores impar sendo enviados para o *Bucket* 1. Podemos então associar cada *bucket* a um único segmento, de forma que apenas a instância responsável pelo segmento possa decidir comandos daquele subconjunto de clientes. Isto evita a duplicação de comandos, já que apenas um dos segmentos poderá decidir comandos de um determinado cliente, e também evita problemas de ordenação, já que, mesmo que um cliente passe a enviar comandos para o líder de outro segmento, este não será capaz de propô-los, já que estes não pertencem ao seu *bucket*.

Figura 20 – Exemplo de encaminhamento de comandos para *buckets*

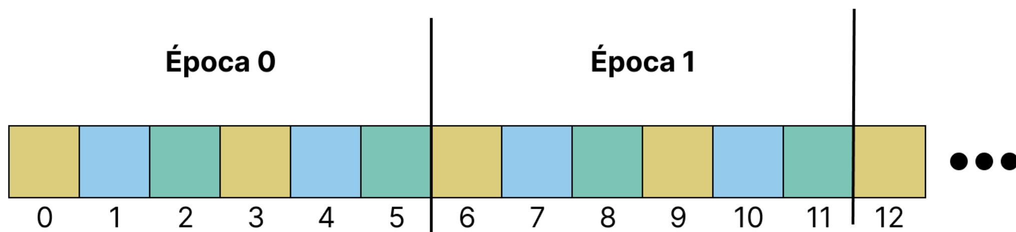


Fonte: De autoria própria.

No entanto, ao permitir que apenas o líder de um determinado segmento possa

decidir sobre os comandos de uma partição de clientes, a falha do líder de um segmento pode levar à indisponibilidade de serviço para todos os clientes associados àquele *bucket*. De forma a lidar com este problema, o ISS introduz o conceito de épocas. Uma época é uma partição do *log* de tamanho n, composta por s segmentos. Diferentemente dos segmentos, que permitem a execução em paralelo, épocas são executadas sequencialmente, ou seja, posições do *log* pertencentes a uma época e somente poderão ser decididas após todos os comandos das épocas 0 até $e - 1$ terem sido decididos. Na Figura 21, podemos ver a partição de um *log* em 2 épocas de tamanho 6, cada uma composta por 3 segmentos de tamanho 2. Neste exemplo, comandos pertencentes à época 1 só poderão ser decididos após os comandos 0 até 5 já terem sido decididos na época anterior.

Figura 21 – Exemplo de encaminhamento de comandos para *buckets*



Fonte: De autoria própria.

Executar as épocas sequencialmente permite que, no momento de troca de épocas, todos os comandos previamente decididos por todos os segmentos sejam conhecidos. Com isso, decisões determinísticas podem ser tomadas sem a necessidade de um coordenador ou de envio de mensagens adicionais entre os nodos. Assim, durante a troca de épocas e antes de iniciar novas instâncias dos algoritmos de consenso, é possível tomar decisões como: reconfiguração do protocolo, mapeamento dos líderes para cada segmento e, especialmente, atribuição de *buckets* para cada segmento, que, se realizada propriamente, garante a atribuição de cada *bucket* para um único segmento ao mesmo tempo que evita os problemas de disponibilidade citados previamente por meio da rotação de *buckets*. Isto também evita a duplicação de comandos em épocas diferentes, pois todos os comandos previamente decididos são conhecidos e todos os comandos da época atual são propostos por um único líder, que não propõe o mesmo comando duas vezes.

4.1 DETALHAMENTO DO ALGORITMO

Com base nestas premissas, detalhamos o comportamento de um nodo executando o *framework* ISS no Algoritmo 1. Ele possui como parâmetros iniciais: o número de *buckets* ($numBuckets$), o tamanho da época ($epochLength$) e o tempo de espera para propor um pedido ($requestTimeout$).

Ao ser inicializado (linha 5), a época atual é definida como 0, criamos uma variável para conter os segmentos ativos ($segments$), definimos que o comando 0 é o próximo a ser executado ($firstUndelivered$) e que nenhum valor já foi entregue ($totalDelivered$). Em seguida, dois vetores são iniciados para conter todos os comandos já decididos (log) e propostos (*proposed*). Todos os *buckets* também são criados e armazenados em um vetor e finalmente a primeira época é iniciada e executada nas linhas 17 e 18. Para executar uma época, o nodo propõe valores para todas as posições desta na qual é um líder (Linha 20).

Além disso, durante a execução da época, o algoritmo aguarda por pedidos de clientes (Linha 27), para então adicionar estes ao seu *bucket* correspondente. Quando um pedido é decidido por um dos segmentos (Linha 30), este é atualizado no *log*, com o pedido sendo removido do seu *bucket* e com comandos previamente propostos diferentes do decidido sendo re-adicionados ao *bucket*. Quando a posição $firstUndelivered$ do *log* é decidida (Linha 39), esta é executada e o valor de $firstUndelivered$ é incrementado. Por fim, quando todas as posições do *log* de uma determinada época e foram decididas (Linha 43), a época $e + 1$ é iniciada.

Iniciamos uma época conforme descrito no Algoritmo 2 ao atualizar o valor da época atual e selecionar o líder de cada segmento por meio de uma política de seleção de líderes (Linha 2), que pode implementar alguma estratégia determinística de reconfiguração em caso de suspeita de falha. Para cada líder selecionado, determinamos seus números de sequência por meio do módulo deste quanto à quantidade de líderes. Depois, seu conjunto de *buckets* é decidido com base na época, no número de segmentos e na quantidade de líderes e, por fim, criamos a instância de *Sequenced Broadcast* responsável pelo segmento e iniciamos sua execução. As funções utilizadas para determinar a qual segmento um determinado número de sequência do *log* pertence e os números de sequência de uma determinada época, são descritos no Algoritmo 3.

Algoritmo 1: Algoritmo principal do ISS para um nodo *p*

```
1  Parameters
2      int(numBuckets)
3      int(epochLength)
4
5  upon init() do
6      currentEpoch ← 0
7      segments ← ∅
8      firstUndelivered ← 0
9      totalDelivered ← 0
10     for sn ∈ ℕ do
11         log[sn] ← none
12         proposed[sn] ← none
13     end for
14     for 0 ≤ b ≤ numBuckets do
15         buckets[b] ← empty bucket
16     end for
17     initEpoch(0)
18     runEpoch(0)
19
20 function runEpoch(e) :
21     for sn ∈ seqNrs(currentEpoch) do
22         if segOf(sn).leader = p then
23             propose(sn)
24         end if
25     end for
26
27 upon event Receive(req) do
28     buckets[hash(req)].add(req)
29
30 upon event Learned(sn, req) do
31     log[sn] ← req
32     if req ≠ none then
33         bucket[hash(req)].remove(req)
34     else if proposed[sn] ≠ none then
35         oldReq ← proposed[sn]
36         buckets[hash(oldReq)].add(oldReq)
37     end if
38
39 upon log[firstUndelivered] ≠ none  do
40     deliver(firstUndelivered)
41     firstUndelivered ← firstUndelivered + 1
42
43 upon ∀sn ∈ seqNrs(currentEpoch) : log[sn] ≠ none do
44     initEpoch(currentEpoch + 1)
45     runEpoch(currentEpoch + 1)
```

Fonte: Adaptado de Stathakopoulou, Pavlovic e Vukolić (2022)

Algoritmo 2: Inicialização de época

```
1  function initEpoch(e) :
2      currentEpoch ← e; leaders ← leader_selection_policy(e) for
        0 ≤ l ≤ |leaders| do
3          seg ← newsegments
4          seg.leader ← leaders[l]
5          seg.seqNrs ← {sn ∈ seqNrs(e)|sn ≡ l (mod |leaders|)}
6          seg.buckets ← Buckets(e, l, leaders)
7          seg.SB ← sb(leaders[l], seg.buckets, seg.seqNrs)
8          seg.init()
9          segments ← segments ∪ {seg}
10     end for
```

Fonte: Adaptado de Stathakopoulou, Pavlovic e Vukolić (2022)

Algoritmo 3: Funções auxiliares no ISS

```
1  function segOf(sn) :
2      return seg ∈ segments : sn ∈ seg.seqNrs
3
4  function seqNrs(e) :
5      return i ∈ ℕ|e · epochLength ≤ i ≤ e · (epochLength + 1)
```

Fonte: Adaptado de Stathakopoulou, Pavlovic e Vukolić (2022)

4.2 IMPLEMENTAÇÃO DO ISS NO PAXI

A partir deste algoritmo, o ISS foi implementado na biblioteca Paxi [1]. Esta seção detalha como esta implementação foi realizada, explorando inicialmente as estruturas de dados criadas de forma a facilitar a execução do algoritmo, em seguida, apresentando a implementação do coordenador, que corresponde ao *framework* ISS, e por fim comentando sobre as adaptações e otimizações realizadas sobre o protocolo Paxos de forma a torná-lo compatível com o ISS.

4.2.1 Estruturas de apoio

Os *buckets*, responsáveis por armazenar os comandos dos clientes, foram implementados como uma lista duplamente encadeada, contendo o pedido do cliente e uma variável lógica indicando se este comando já foi decidido. Além disso, cada *bucket* também possui um *mutex*, utilizado para evitar acessos simultâneos, e um dicionário chave-valor, que mapeia cada comando ao seu item na lista, de forma a reduzir a complexidade de acesso aos itens. Cada *bucket* possui como funções:

[1] `https://github.com/g-buzzi/iss-paxos`

- **Add:** Adiciona o comando na lista encadeada e, caso ainda não esteja presente, o inclui no dicionário chave-valor;
- **Commit:** Remove o comando da lista encadeada e atualiza o dicionário chave-valor para indicar que o pedido já foi decidido;
- **Get:** Retorna o primeiro item da lista ou *nil* caso nenhum item exista.

Além dos *buckets*, também foram implementados grupos de *buckets*, uma estrutura responsável por agrupar todos os *buckets* pertencentes a um segmento. A principal função deste grupo é selecionar os pedidos que serão propostos pelo líder do segmento. Para isso, um grupo de *buckets* possui como atributos uma lista de *buckets*, um temporizador para aguardar por pedidos e um canal utilizado pelos *buckets* para indicar quando um item for adicionado.

O grupo de *buckets* implementa a função *get*, chamada pelo líder do segmento quando este precisa propor um comando. A função, descrita na linha 6 do Algoritmo 4, tenta selecionar um comando em algum de seus *buckets* por meio da função *selectRequest*. Caso o valor retornado não seja vazio ou se o grupo estiver configurado para não aguardar por comandos (Linha 8), este valor é retornado.

Já caso não exista nenhum pedido e o algoritmo tenha sido configurado para aguardar por comandos, um comando guardado (DIJKSTRA, 1975), que executa o código da primeira condição a ser verdadeira, é utilizado na linha 11. Além do comando guardado, uma estrutura de canal também é utilizada. Esta estrutura, inicializada pelo comando **chan** c, é capaz de enviar um determinado valor v para o canal c ($c\mathbf{!}\,v$), que então pode ser recebido por meio do mesmo canal c com uma operação bloqueante ($c\mathbf{?}\,v$), que aguarda até que um valor v seja recebido antes de prosseguir com a execução. Com esses canais, quando um *bucket* sinaliza a inclusão de um comando (*requestAdded*), este comando é retornado. Do contrário, um valor *none* é retornado quando um sinal indicando que o tempo de espera excedeu o seu limite for recebido.

Para selecionar pedidos de forma justa, a função *selectRequest* foi implementada (Linha 17). Nesta, os *buckets* são percorridos, a partir do último a ter sido verificado (*lastBucket*). Quando um *bucket* com comandos é encontrado, atualizamos a variável global *lastBucket*, de forma a continuar a verificar a partir deste número na próxima seleção de pedidos, e um comando é retornado (Linha 23). Caso nenhum *bucket* possua comandos, um valor vazio é retornado (Linha 26).

Algoritmo 4: Obter pedido a partir de grupo de *buckets*

```
1  Parameters
2      chan requestTimeout
3      Bucket[] buckets
4      int lastBucket
5
6  function get() :
7      req = selectRequest()
8      if req ≠ none ∨ requestTimeout = 0  then
9          return req
10     end if
11     select :
12         case requestAdded? r :
13             return selectRequest()
14         case requestTimeout? t :
15             return none
16
17 function selectRequest() :
18     for 0 ≤ b ≤ |buckets| do
19         bucketNum ← b + lastBucket + 1 (mod |buckets|)
20         bucket ← buckets[bucketNum]
21         if bucket.nReqs > 0 then
22             lastBucket ← bucketNum
23             return bucket.get()
24         end if
25     end for
26     return none
```

Fonte: Autoria própria

4.2.2 Coordenador

Utilizando-se destas estruturas de apoio, o coordenador, representado pela classe ISS, foi implementado. Este é responsável pelo gerenciamento das épocas, desde sua criação até seu término, a execução dos comandos decididos e o recebimento e correto tratamento de mensagens encaminhadas ao nodo. O algoritmo como um todo possui como parâmetros ajustáveis, o número de *buckets*, tamanhos dos segmentos e quantidade de segmentos a serem executados. Além destes parâmetros ajustáveis, a classe ISS é iniciada com as seguintes informações:

- ***Node:*** Todas as informações do nodo no qual ele está hospedado;
- ***config:*** uma lista contendo o identificador de todos os nodos em ordem numérica;
- ***log:*** um dicionário chave-valor, que mapeia posições do log às entradas do log;
- ***currentEpoch:*** um valor inteiro que representa a época sendo atualmente executada por este nodo;
- ***segments**:* uma lista contendo ponteiros para todas as instâncias de segmentos da época atual;
- ***epochBuffer:*** uma pilha de mensagens que devem ser entregues para segmentos de uma época futura;
- ***buckets:*** uma lista contendo todos os *buckets*;
- ***execute:*** um número inteiro que representa a próxima posição do *log* a ser executada;
- ***logChan:*** um canal da linguagem Go por onde atualizações no *log* são enviadas por um segmento caso um comando seja decidido;
- ***messageChan:*** um canal onde mensagens endereçadas para os segmentos são adicionadas;
- ***bufferChan:*** um canal por onde uma mensagem é enviada para ser adicionada no *epochBuffer*;
- ***retrieveBuffer:*** um canal por onde um valor lógico é enviado quando mensagens presentes no *buffer* devem ser recuperadas;
- ***epochEnd:*** um canal por onde um valor lógico é enviado quando a época chegou ao seu fim.

A partir destes valores, de forma a evitar problemas de concorrência, o ISS executa paralelamente 4 rotinas, responsáveis por: tratar mensagens de clientes (*handleRequest*),

gerenciar a execução das épocas e mensagens internas ($runEpoch$), gerenciar o *buffer* de mensagens ($bufferManager$) e atualizar o *log* e executar comandos ($exec$). As mensagens de cliente possuem um simples tratamento, descritas no Algoritmo 5, onde passam por uma função *hash* para determinar o *bucket* correto e então são adicionadas a este *bucket*.

Algoritmo 5: Co-rotina *handleRequest*

```
1  Parameters
2      Bucket[] buckets
3      int numBuckets
4
5  function handleRequest(r) :
6      bucketID ← hash(r.clientID)
7      buckets[bucketID].add(r)
8
9  function hash(clientID) :
10     return clientID (mod numBuckets)
```

Fonte: Autoria própria

Já a sub-rotina *runEpoch* é a responsável por iniciar novas épocas e realizar o tratamento de mensagens recebidas e direcionadas para um determinado segmento em uma época. Ela é a única co-rotina capaz de atualizar a época atual, de forma a evitar que mensagens de um segmento da época e sejam incorretamente enviadas para um segmento da época $e + 1$.

Conforme descrito no Algoritmo 6, a sub-rotina inicia a época 0 e, então, inicia um laço de repetição indefinido, onde, por meio de um comando guardado, aguarda por um de dois eventos: o fim da época atual ou o recebimento de uma mensagem interna. No caso do fim da época atual (Linha 11), todos os segmentos atuais recebem uma mensagem especial para finalizar sua execução, a próxima época é iniciada e é enviado um sinal para recuperar as mensagens do *buffer*. Já no caso de recebimento de uma mensagem interna (Linha 17), caso esta seja de uma época atual, ela é enviada para o segmento correspondente, e, caso seja de uma época futura, ela é enviada para o *buffer* de mensagens.

O *buffer* de mensagens é o responsável por armazenar todas as mensagens de épocas futuras e, no momento de troca de épocas, reviver as mensagens relevantes. Para isso, conforme descrito no Algoritmo 7, o gerenciador de *buffer* executa um laço de repetição indefinido, onde, por meio de um comando guardado, aguarda por uma mensagem para ser adicionada ou um sinal que as mensagens devem ser recuperadas.

Quando uma mensagem precisa ser adicionada no *buffer* (Linha 10), caso a época desta seja atual, ela é encaminhada para o canal de mensagens, e caso seja futura ela é adicionada no *buffer*. Já durante a recuperação de mensagens (Linha 16), uma nova pilha ($futureStack$) é criada para armazenar as mensagens futuras. Todas as mensagens da pilha atual são então removidas desta e verificadas. Caso a mensagem seja da época atual, ela é enviada para o canal de mensagens ($messageChan$). Já mensagens de épocas

Algoritmo 6: Co-rotina *runEpoch*

```
1  Parameters
2      int currentEpoch = 0
3      int numSegments
4      Segment[] segments
5      chan epochEnd
6
7  function runEpoch() :
8      startEpoch(0)
9      while true do
10         select :
11             case epochEnd? :
12                 for 0 ≤ i < numSegments do
13                     segments[i].workQueue ! < End >
14                 end for
15                 startEpoch(currentEpoch + 1)
16                 retrieveBuffer! true
17             case messageChan? m :
18                 if m.epoch = currentEpoch then
19                     segments[m.segment].workQueue! m
20                 else if m.epoch > currentEpoch then
21                     bufferChan! m
22                 end if
23     end
```

Fonte: Autoria própria

futuras são adicionadas em $futureStack$. Após todas as mensagens da pilha terem sido processadas, a pilha é substituída por $futureStack$.

Por fim, o preenchimento do *log* e entrega de comandos é descrito no Algoritmo 8. Este executa um laço de repetição indeterminado, que aguarda por atualizações ao *log*, enviadas pelos segmentos por meio do canal *logChan*. A cada atualização, o algoritmo atualiza seu *log* e obtém a entrada na próxima posição a ser executada (*execute*). Caso esta não esteja vazia, o comando é entregue, *execute* é incrementado e o próximo comando é verificado. Ao entregar um comando, também se verifica se a época atual já chegou ao fim, sendo que, neste caso, é enviado um sinal para o canal *epochEnd*, indicando que a execução da próxima época deve ser iniciada.

Algoritmo 7: Gerenciador de *buffer*

```
1  Parameters
2      stack epochBuffer
3      int currentEpoch
4      chan retrieveBuffer
5      chan messageChan
6
7  function bufferManager() :
8      while true do
9          select :
10             case bufferChan? m  :
11                 if m.epoch = currentEpoch then
12                     messageChan! m
13                 else if m.epoch > currentEpoch then
14                     epochBuffer.add(m)
15                 end if
16             case retrieveBuffer? e  :
17                 futureStack ← newStack
18                 for 0 ≤ i < epochBuffer.length do
19                     m ← epochBuffer.pop()
20                     if m.epoch = currentEpoch then
21                         messageChan! m
22                     else if m.epoch > currentEpoch then
23                         futureStack.add(m)
24                     end if
25                 end for
26                 epochBuffer ← futureStack
27     end
```

Fonte: Autoria própria

Algoritmo 8: Execução de comandos

```
1  Parameters
2      LogItem[] log
3      chan logChan
4      int execute
5      int epochSize
6      int currentEpoch
7      chan epochEnd
8
9  function exec() :
10     for true do
11         logChan? update
12         log[update.slot] ← update.entry
13         e ← log[execute]
14         for e ≠ none do
15             deliver(log[execute])
16             execute ← execute + 1
17             e ← log[execute]
18             if execute = (currentEpoch + 1) · epochSize then
19                 epochEnd! true
20             end if
21         end for
22     end for
```

Fonte: Autoria própria

4.2.3 Adaptações ao protocolo Paxos

De forma a aplicar o *framework* ISS sobre o algoritmo Paxos original, foram necessárias adaptações de forma que este se comporte como uma instância de *Sequenced Broadcast*. Estas adaptações encontram-se descritas no Algoritmo 9 e **??**. Em primeiro lugar, ao ser iniciado (Linha 11 do Algoritmo 9), um temporizador é começado e duas co-rotinas são executadas, a co-rotina *worker* e $heartbeatMonitor$. Além disso, o líder inicial do segmento também passa a propor comandos ativamente, propondo valores de seus grupos de *buckets* até que todas as posições do *log* estejam preenchidas, de forma a garantir que as épocas avancem em um ritmo constante.

A co-rotina *worker* (Linha 20 do Algoritmo 9) foi criada para evitar problemas de concorrência devido a chamadas de funções provenientes de diferentes co-rotinas. Ela implementa uma fila de trabalho que serializa a execução de funções de tratamento e envio de mensagens. Para isso, é realizado um laço de repetição onde funções de tratamentos são chamadas para trabalhos (job) recebidas por meio do canal *workQueue*. Quando recebe uma mensagem do tipo $< End >$, o trabalhador finaliza a execução de mensagens e envia um sinal -1 para o canal *heartbeat*, de forma que este saiba que a execução do segmento chegou ao fim.

Este sinal é destinado à co-rotina $heartBeatMonitor$, descrita na Linha 6 do Algoritmo 10, que age como o detector de falhas do algoritmo, ao aguardar por sinais no canal *heartbeat*. Sinais 0 indicam que uma mensagem foi processada e o algoritmo está sendo executado normalmente, sendo que possuem como única função reiniciar o temporizador. Os sinais -1 são enviados pelo *worker* e indicam que o segmento chegou ao fim e que o detector de falhas pode finalizar sua execução. Por fim, sinais 1 são enviados pelo temporizador (Linha **??** do Algoritmo 10), que envia este sinal quando o tempo estipulado for ultrapassado, de forma que o nodo possa enviar uma mensagem 1A, em uma tentativa de se tornar o líder do segmento e avançar para a época seguinte.

Caso seja eleito (Linha 17 do Algoritmo 10), o algoritmo verifica todas as posições do *log* do segmento, enviando mensagens SKIP ($< P2A, \mathit{none} >$) para posições vazias do *log* (Linha 20) e tentando decidir comandos previamente propostos e ainda não decididos (Linha 22).

Por fim, foi possível otimizar o número de mensagens necessárias para decidir um comando vazio (Mensagens *SKIP*) devido às propriedades do *Sequenced Broadcast*. Assim como no Mencius, como o SB limita os valores que líderes diferentes do inicial podem propor, quando um nodo, seja ele o líder inicial ou não, propõe um comando vazio, este já pode ser considerado decidido pelo algoritmo apenas com a mensagem 2A. Isto vem do fato de:

- Caso o líder inicial esteja propondo um valor vazio, como ele era o único que poderia propor um valor diferente, temos a garantia que qualquer nodo assumindo a liderança

Algoritmo 9: Adaptações ao Paxos

```
1  Parameters
2      bucketGroup bucketGroup
3      bool starterLeader
4      bool activeLeader
5      int segmentSize
6      chan workQueue
7      chan heartbeat
8      timer timer
9      LogItem[] segmentLog
10
11 upon init do
12     timer.start()
13     run worker() run heartbeatMonitor()
14     if starterLeader then
15         for (0 ≤ i < segmentSize) ∧ activeLeader do
16             request ← bucketGroup.get()
17             workQueue! <P2A, request>
18         end for
19     end if
20 function worker() :
21     keepAlive ← true
22     while keepAlive do
23         workQueue? job
24         switch job.type do
25             case <End> :
26                 keepWorker ← false
27             Other :
28                 handle(job)
29         end
30     end
31     heartbeat! -1
```

Fonte: Autoria própria

no futuro também proporia o valor vazio.

- Caso seja um líder eleito, este irá propor um valor previamente proposto ou vazio. Caso proponha vazio, assim como no caso anterior, nenhum outro nodo seria capaz de propor um valor diferente em rodadas futuras, já que a rodada inicial, onde valores diferentes podem ser propostos, já chegou a um fim.

Algoritmo 10: Adaptações ao Paxos - Detector de falhas

```
1  Parameters
2      chan workQueue
3      chan heartbeat
4      timer timer
5
6  function heartbeatMonitor() :
7      status ← 0
8      while status ≥ 0 do
9          heartbeat? status
10         if status = 1 then
11             workQueue! <P1A>
12         end if
13         timer.reset()
14     end
15 upon timeout  do
16     heartbeat! 1
17 upon elected  do
18     for 0 ≤ s < segmentSize do
19         if segmentLog[s] = none then
20             workQueue! <P2A, none>
21         else if segmentLog[s].commit = false then
22             workQueue! <P2A, segmentLog[s].request>
23     end for
```

Fonte: Autoria própria

5 AVALIAÇÃO EXPERIMENTAL

Com o objetivo de mensurar a vazão, latência e escalabilidade do protocolo de consenso implementado, assim como o efeito de alguns parâmetros em seu desempenho, foram realizados experimentos de forma a simular a execução de um sistema cliente-servidor replicado por este. Os dados de vazão e latência foram coletados por meio do gerador de carga inerente à biblioteca Paxi e, de forma a avaliar a escalabilidade dos protocolos, foram realizados experimentos com 3 e 6 nodos, de forma a comparar os efeitos da adição de novos nodos no algoritmo. Para avaliar o desempenho do algoritmo implementado em comparação a outros algoritmos de consenso multi-líder, todos os experimentos citados também foram realizados com os protocolos WPaxos e EPaxos, que representam duas estratégias diferentes de aumento de vazão por meio do acréscimo do número de líderes simultâneos.

5.1 CONFIGURAÇÃO

Os experimentos foram realizados na plataforma Emulab (WHITE *et al.*, 2002), um ambiente *on-line* que disponibiliza uma variedade de computadores físicos e virtuais sob-demanda para a realização de pesquisas em redes e sistemas distribuídos. Para os experimentos, foram utilizados nodos físicos do tipo d430, cujas configurações estão descritas na Tabela 1. Estes nodos foram organizados em uma estrutura de rede que simula uma *Local Area Network* (LAN), com todos os nodos podendo se comunicar entre si por meio desta rede.

Tabela 1 – Configurações de cada Nodo

Componente	**Especificação**
Processador	Intel Xeon E5-2630 v3
Núcleos	8
Threads por núcleo	2
Memória RAM	65GB
Placa de rede	1Gbps
Sistema Operacional	Ubuntu 20.04 64 bits
Versão GoLang	1.22.1

Fonte: De autoria própria.

De forma a evitar variações nos resultados causadas por divisão de recursos, cada uma das máquinas simulou exclusivamente um agente do protocolo de consenso, ou seja, simulou exatamente 1 cliente ou 1 servidor. Em todos os experimentos, 6 máquinas foram alocadas para atuar como clientes, executando o gerador de carga da biblioteca Paxi.

As configurações dos nodos clientes encontra-se na Tabela 2, onde podemos ver que cada teste de carga foi realizado por 1 minuto, durante o qual cada servidor simulou um banco de dados chave-valor com 12000 chaves. Cada um dos nodos clientes simulou

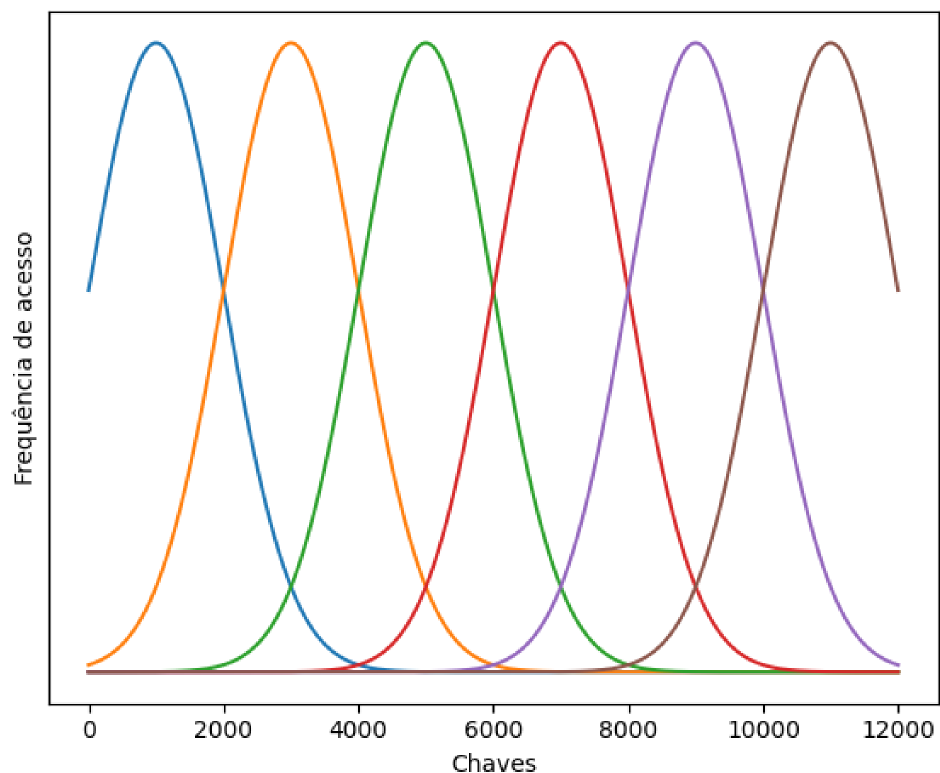
256 clientes concorrentes, que realizavam apenas leituras, de forma a evitar colisões que poderiam reduzir a vazão do EPaxos. Por fim, de forma a simular um comportamento de acesso onde cada cliente é mais provável de acessar um subconjunto de chave do que outro, cada cliente utilizou uma distribuição normal, definida por σ e μ, para determinar a probabilidade de acessar uma determinada chave. A forma da distribuição normal de todos os clientes pode ser vista na Figura 22, onde cada curva normal indica a frequência de acesso a determinada chave para um nodo cliente diferente.

Tabela 2 – Parâmetros do gerador de carga

Parâmetro	**Valor**
Tempo (T)	60 segundos
Número de chaves (K)	12000
Porcentagem de escritas (W)	0%
Concorrência	256
Distribuição	normal
Média da distribuição (μ)	$(n \cdot 2000) - 1000$
Desvio padrão da distribuição (σ)	1000

Fonte: De autoria própria.

Figura 22 – Distribuição do acesso às chaves



Fonte: De autoria própria.

Também é a partir destes 6 nodos clientes que os dados de vazão e latência são obtidos. A vazão, em pedidos por segundo, é calculada como a soma do número total de respostas que todos os nodos clientes receberam em um determinado segundo. Já a latência é calculada como a diferença entre o tempo de envio do pedido e o recebimento de uma resposta para todos os pedidos iniciados em um determinado segundo.

Por fim, de forma a melhor entender os efeitos dos parâmetros do *framework* ISS no desempenho deste, o valor de tamanho de segmento e tempo de espera por pedidos foi variado, conforme apresentado na Tabela 3. Espera-se que o aumento do tamanho dos segmentos leve a uma vazão mais elevada, devido a uma maior paralelização entre segmentos e redução do impacto da sobrecarga de inicialização de épocas, devido ao maior tamanho da época. Quanto ao tempo de espera por um determinado pedido, supõe-se que sua existência pode reduzir a quantidade de mensagens vazias sendo enviadas, reduzindo assim a latência. Além disso, é importante destacar que os primeiros experimentos foram realizados sem a otimização das mensagens *SKIP* citadas no capítulo anterior, com os resultados desta otimização sendo abordados posteriormente.

Tabela 3 – Parâmetros avaliados para o ISS

Nome	**Tamanho do segmento**	**Tempo de espera por pedido (ms)**
ISS_32s_0t	32	0
ISS_32s_30t	32	30
ISS_64s_0t	64	0
ISS_64s_30t	64	30
ISS_128s_0t	128	0
ISS_128s_30t	128	30

Fonte: De autoria própria.

5.2 MÉTRICAS DE AVALIAÇÃO

Para todos os experimentos, cada um dos algoritmos avaliados foi executado uma vez para cada configuração do regulador (*throttle*) por um total de 60 segundos. A vazão de cada variação do regulador foi calculada como a média do número de comandos executados pelo sistema a cada segundo de execução. A vazão máxima de cada protocolo foi calculada como o máximo valor dentre as vazões médias de todas as configurações do regulador para um determinado algoritmo.

As métricas de latência mediana e 99 percentil foram obtidas a partir, respectivamente, da mediana e do 99º percentil do tempo necessário para o recebimento de uma resposta para cada um dos pedidos enviados em uma determinada configuração do regulador. A mediana foi selecionada de forma a demonstrar o comportamento que será experienciado pela maioria dos clientes, sendo escolhida em vez da latência média por não ser tão afetada pelos valores de cauda. A latência 99 percentil foi selecionada para

representar as piores latências experienciadas pelos clientes, não levando em consideração possíveis valores discrepantes.

5.3 EXPERIMENTOS COM 3 NODOS

Durante os experimentos com 3 nodos, o comportamento padrão dos clientes que enviam as mensagens para os nodos do ISS foi alterado. De forma a manter todos os *buckets* com pedidos, ao invés de enviar todos os seus pedidos para um único servidor, estes seriam enviados para servidores de forma aleatória.

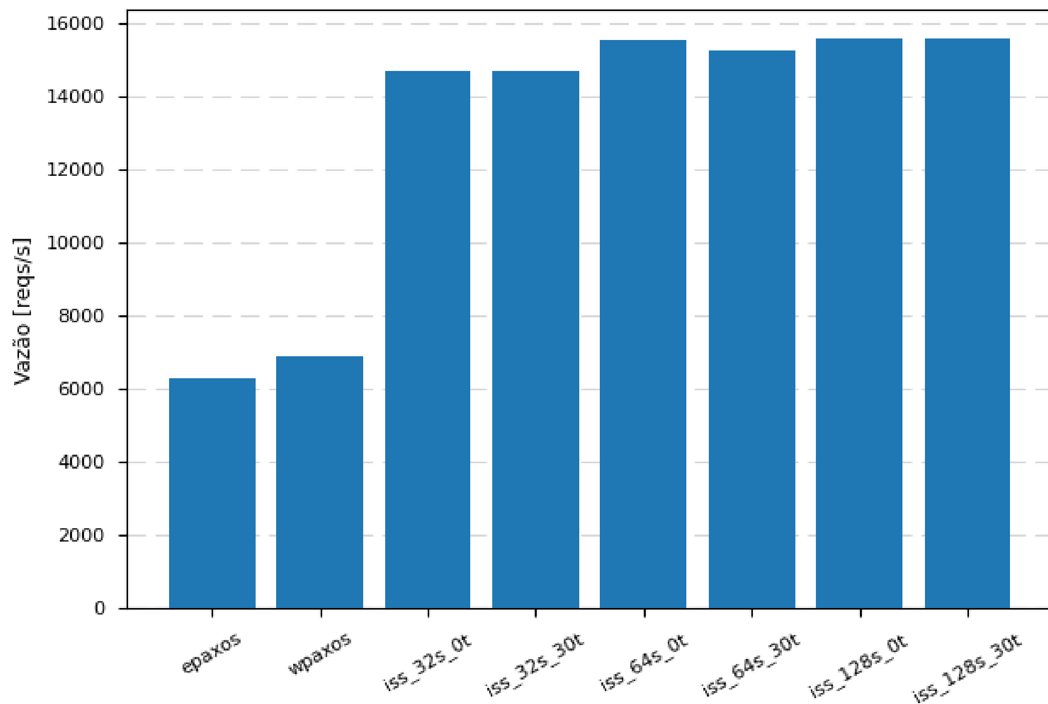
Quanto aos resultados obtidos, podemos ver na Figura 23 que, com estas configurações, a vazão do *framework* implementado é mais do que o dobro daquela do EPaxos e WPaxos. O maior ganho de vazão do ISS em relação ao WPaxos e EPaxos provavelmente provem do paralelismo de sua execução, onde, além de permitir que múltiplos nodos enviem propostas simultaneamente como nos outros algoritmos, também paraleliza a execução do algoritmo em si, com uma co-rotina responsável por cada um dos seus segmentos. Com isso, o ISS consegue ter um melhor aproveitamento dos núcleos e *threads* de cada máquina e atingir uma maior vazão antes de sua saturação.

Quanto aos efeitos dos parâmetros do ISS em si, podemos observar um leve crescimento na vazão máxima do algoritmo com o aumento do tamanho dos segmentos. Isto ocorre pois, com segmentos maiores, uma maior quantidade de pedidos pode ser enviada por um nodo antes deste precisar aguardar que os outros nodos preencham seus segmentos e enviem suas mensagens de *commit*, tal que uma maior quantidade de pedidos se encontram em tráfego simultaneamente. Quanto ao parâmetro de tempo de espera, este não teve grandes efeitos na vazão, principalmente porque em situações de alta vazão sempre existem pedidos disponíveis para serem propostos e o algoritmo nunca espera por um pedido.

Já quanto a latência, conforme pode ser visto na Figura 24, o WPaxos e EPaxos apresentaram latências medianas próximas de 1ms quando processando 5400 pedidos por segundo. Já o ISS apresentou latências muito mais elevadas, inicialmente entre os 14 e 60 milissegundos, devido ao fato de, além do tempo necessário para a tomada de uma decisão, clientes que encaminham pedidos para nodos que ainda não são líderes de seu *bucket* também precisam aguardar até que n épocas terminem e o nodo finalmente possa propor o comando deste cliente. Caso o nodo esteja configurado para aguardar por pedidos, esta característica é ainda mais evidente, como no iss_32s_30t, onde temos um aumento na latência mediana devido ao fato de cada época durar mais tempo.

Também é possível notar, devido a esta característica, que instâncias do ISS com segmentos menores também apresentaram menor latência mediana, devido ao fato de suas épocas chegarem ao fim mais frequentemente, reduzindo o tempo de espera até que um determinado nodo possa propor pedidos de um *bucket* específico. Em combinação com o observado quanto à vazão, podemos concluir que o aumento do tamanho do segmento,

Figura 23 – Vazão máxima (3 nodos)



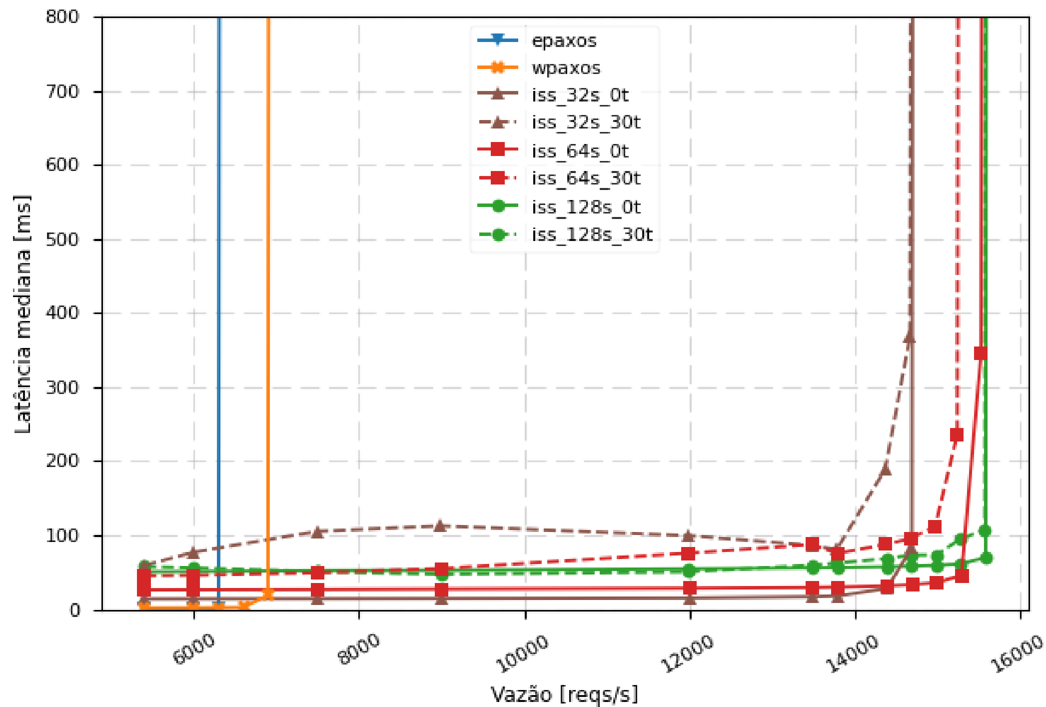
Fonte: De autoria própria.

apesar de aumentar a vazão do algoritmo, também aumenta sua latência em situações de baixa vazão.

Também podemos ver na Figura 25, nos nodos sem tempo de espera, um padrão similar ao citado até o momento. Nodos com segmentos menores apresentam menor latência em situações de baixa latência, mas são capazes de lidar com uma menor quantidade total de pedidos por segundo.

No entanto, o comportamento de nodos que aguardam por pedidos se mostrou praticamente aleatório. Isto foi causado por uma má distribuição dos pedidos nos *buckets* dos nodos, com alguns nodos recebendo mais ou menos pedidos de um determinado cliente, de forma que estes não foram atendidos em velocidades consistentes. De forma a evitar esta distorção, para experimentos com 6 nodos, o comportamento dos clientes foi alterado. Agora, estes passam a enviar pedidos para um único nodo, com estes pedidos sendo enviados para *buckets* não com base no identificador do cliente, mas sim em seu número de pedido. Este método de particionar os pedidos não garante que os pedidos serão ordenados corretamente entre si no caso do envio destes para nodos diferentes, mas foi utilizado por simular uma carga mais balanceada, que seria esperada em sistemas com um maior número de clientes.

Figura 24 – Latência Mediana (3 nodos)



Fonte: De autoria própria.

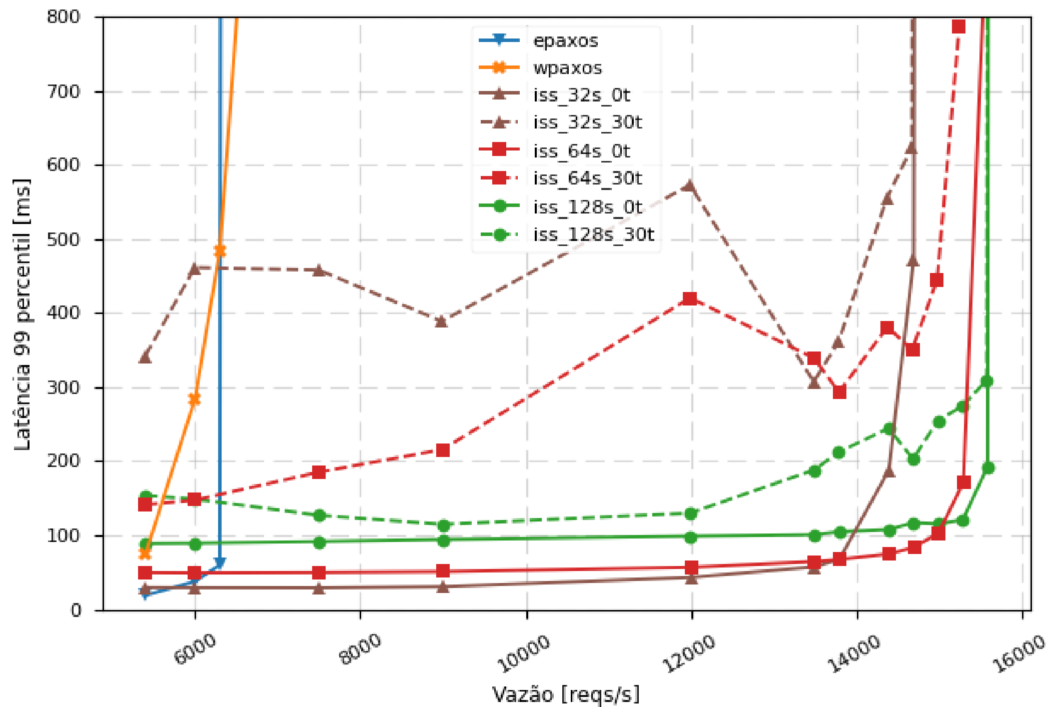
5.4 EXPERIMENTOS COM 6 NODOS

Ao aumentar o número de nodos, tivemos uma redução na vazão de todos os algoritmos, conforme pode ser visto na Figura 26, onde comparamos a vazão de cada configuração. A maior redução relativa de vazão foi no EPaxos, com uma redução em aproximadamente 14% de sua vazão máxima. Em seguida, temos, independente da configuração, o ISS, que apresentou uma redução de vazão próxima dos 10%, e com a menor redução relativa temos o WPaxos, que perdeu apenas 4% de sua vazão com o aumento de nodos.

Esta métrica pode indicar que o WPaxos apresenta uma melhor escalabilidade do que a versão do ISS implementada, podendo apresentar resultados melhores do que esta em situações com um número elevado de nodos. No entanto, também é possível que a maior redução percentual de vazão no ISS seja devido a uma diferença absoluta na quantidade de mensagens que este estava enviando em comparação aos outros protocolos. Neste caso, ainda é possível que este siga com uma vazão comparável ou até maior que a do WPaxos, mesmo em situações com uma grande quantidade de réplicas.

Quanto a latência, o EPaxos e WPaxos tiveram um aumento relativamente pequeno, não passando de 2 milissegundos ao processar 5400 pedidos por segundos. Já no ISS, para os nodos que não aguardam por pedidos, as latências praticamente quadruplicaram em relação à sua latência mediana com 3 nodos. Isto ocorre por diversos motivos, dentre eles,

Figura 25 – Latência 99 percentil (3 nodos)



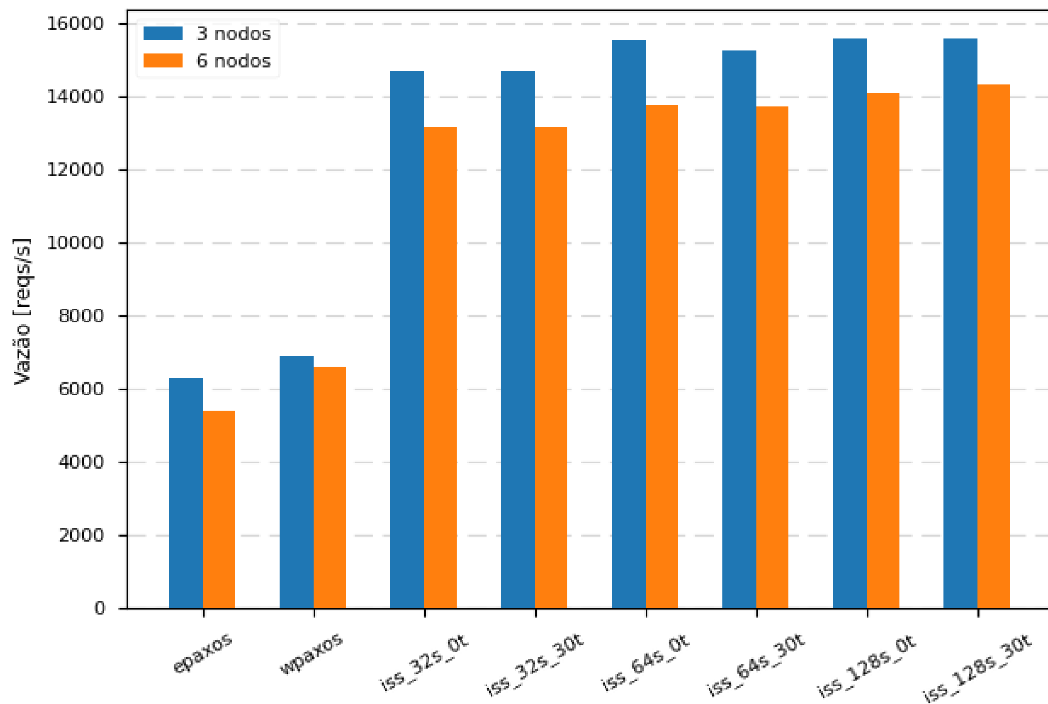
Fonte: De autoria própria.

o aumento do tamanho da época devido ao aumento no número de segmentos (de 3 para 6), que aumenta a latência mediana devido a espera até o nodo se tornar líder de um novo *bucket*, e o aumento do número de mensagens *SKIP* sendo enviadas e que utilizam tempo de processamento que poderia ser utilizado para decidir pedidos reais.

Outra consideração é que, com uma carga mais balanceada de clientes, podemos observar o comportamento do ISS quando este espera por pedidos em um ambiente balanceado. Podemos ver que, em comparação a configurações que não esperam, estas começam com uma latência mediana superior, devido ao fato de demorarem para avançar de época. Já em situações de vazão mediana, podem apresentar latências similares ou até melhores do que os algoritmos que não esperam, já que não desperdiçam tantos recursos com mensagens *SKIP*. E em situações próximas de seu ponto de saturação, possuem latências similares a de algoritmos que esperam, devido ao fato de não ser mais necessário esperar por pedidos.

Na Figura 28, que mostra a latência 99 percentil, podemos ver o mesmo padrão observado anteriormente, com o EPaxos e WPaxos mantendo latências relativamente baixas, e as latências do ISS sem espera sendo praticamente quadruplicadas. O ISS com espera também apresenta a mesma parábola de latência que anteriormente. No entanto podemos notar que, apesar de sua vazão mediana chegar próxima das configurações sem espera, sua latência 99 percentil é quase sempre maior.

Figura 26 – Comparação de Vazão (3x6 nodos)



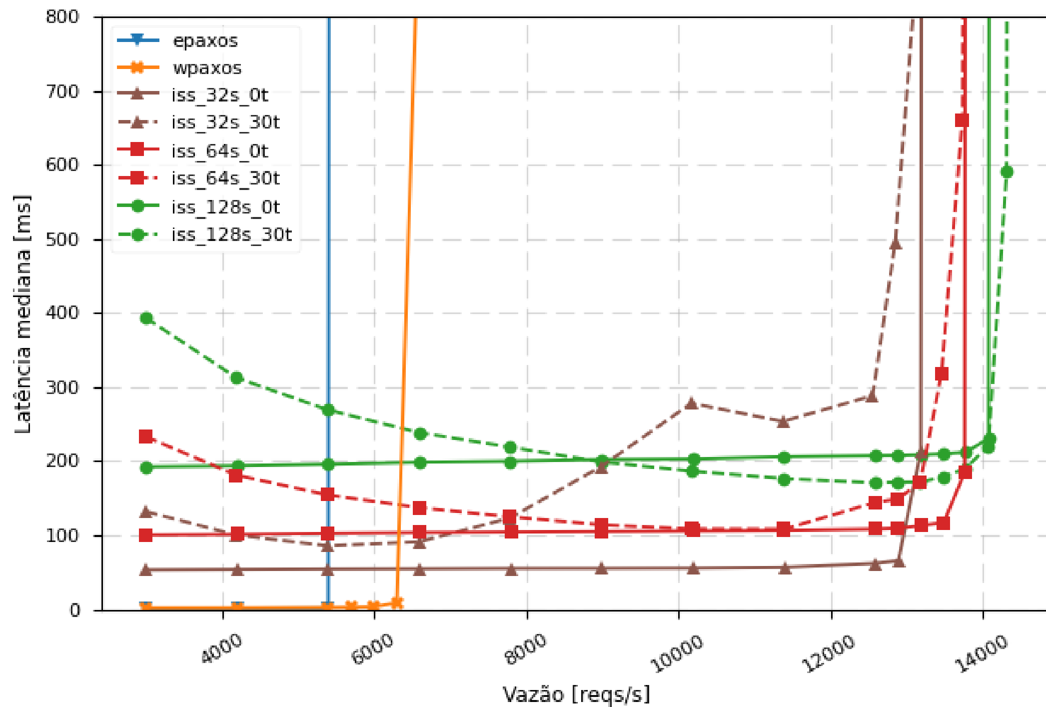
Fonte: De autoria própria.

5.5 OTIMIZAÇÃO DAS MENSAGENS *SKIP*

Por fim, de forma a reduzir os custos de decidir um valor vazio, foi aplicada a otimização da mensagem *SKIP* mencionada no Capítulo **??**, onde comandos vazios podem ser rapidamente decididos com uma só mensagem. Esta otimização, conforme pode ser vista na Figura 29, não trouxe grande impacto na vazão das configurações, já que, em situações próximas da saturação do algoritmo este não envia mensagens *SKIP*.

No entanto, em situações de baixa vazão, podemos perceber que ocorre uma grande redução na latência para os nodos que não esperam por pedidos, conforme pode ser observado na Figura 30. Para estes nodos que não aguardam por pedidos, e então enviam mensagens *SKIP* mais frequentemente, a latência mediana caiu pela metade quando lidando com 5400 pedidos por segundo. Também podemos notar um gradual aumento na latência conforme o número de requisições processadas aumenta, causada pela diminuição do envio de mensagens *SKIP* e maior proporção de comandos reais sendo decididos. Assim diferentemente do algoritmo não otimizado que, para segmentos de tamanho 64 mantinha uma latência mediana quase constante de 100ms, o algoritmo otimizado possui uma latência inicial próxima dos 50ms, chegando aos 100ms apenas em situações de alta vazão. Já as configurações do ISS que esperam por comandos não foram muito afetadas pela otimização, já que enviam uma quantidade muito menor de mensagens *SKIP*.

Figura 27 – Latência Mediana (6 nodos)



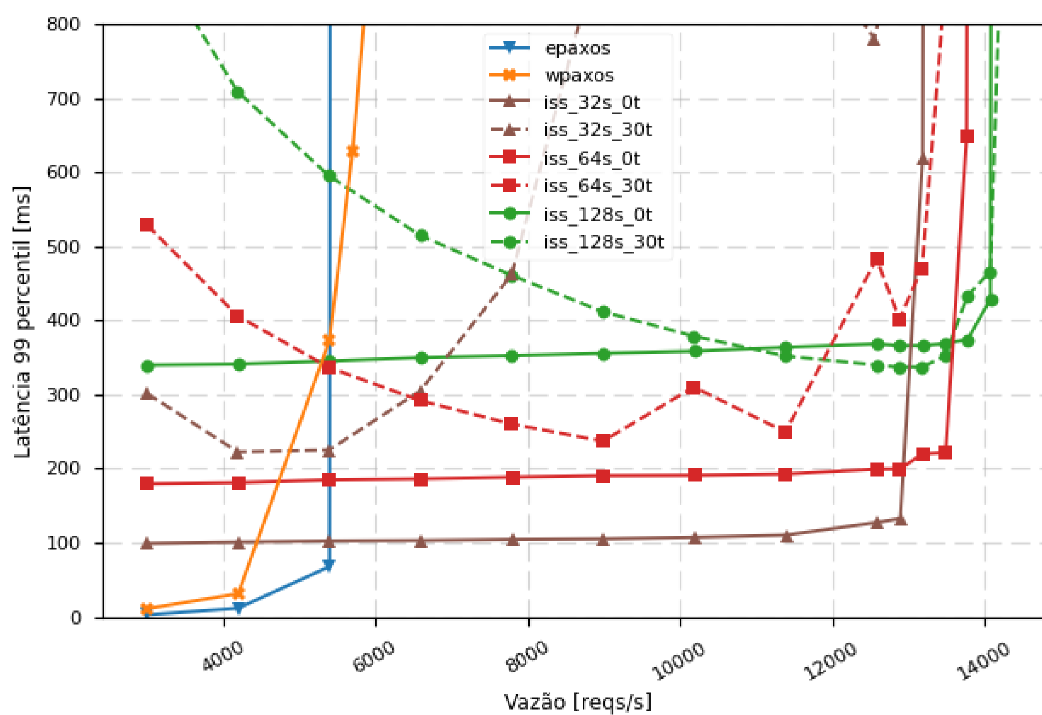
Fonte: De autoria própria.

Também vale destacar que o mesmo padrão de latência observado na latência mediana pode ser visto na Figura 31, com a latência em situações de baixa vazão sendo reduzida praticamente pela metade em nodos que não esperam por pedidos. Novamente, a otimização da mensagem *SKIP* não trouxe grandes variações na latência.

A partir destes resultados, podemos concluir que, apesar do *framework* implementado apresentar vazão superior aos outros quando replicando 3 ou 6 nodos, este apresenta uma métrica de escalabilidade pior que a do WPaxos, sendo que potencialmente não escalaria tão bem quanto este. Quanto a latência, mesmo com a otimização das mensagens *SKIP*, o ISS apresentou latências elevadas devido ao fato de ser necessário esperar até que o nodo que recebeu o pedido se torne líder do *bucket* correto para então o propor.

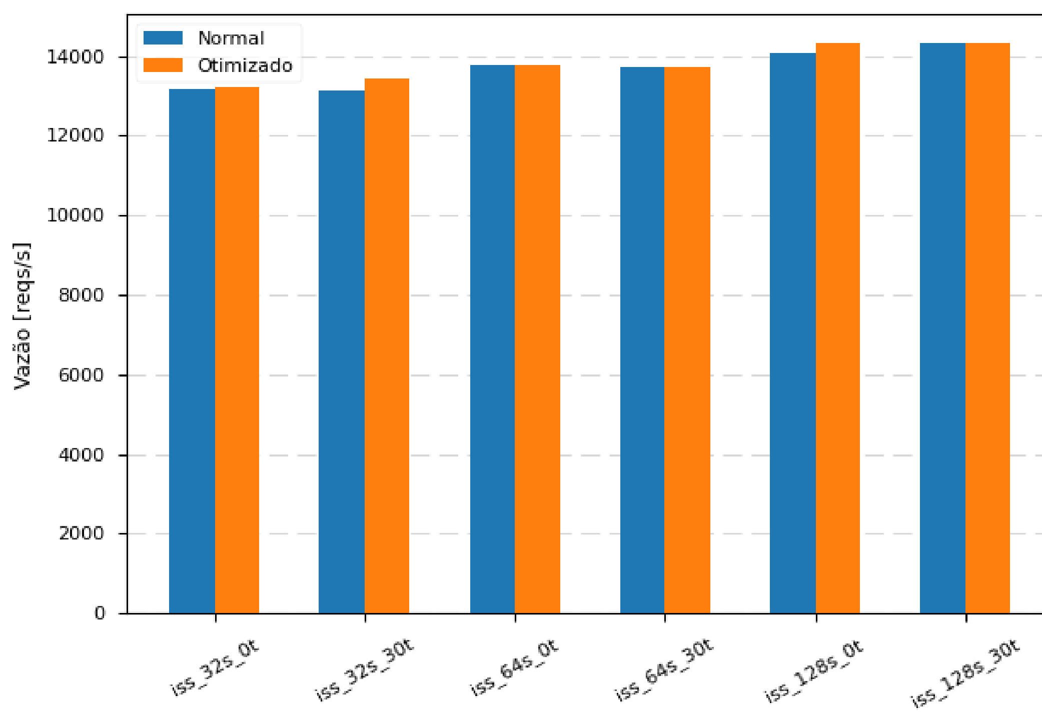
Apesar disso, trabalhos futuros poderiam melhorar a implementação do ISS por meio da utilização de quorums flexíveis, que se tornariam ainda mais relevantes com o aumento da quantidade de nodos, e com estratégias de *batching*, conhecidas por possibilitar o aumento da vazão de algoritmos de consenso ao permitir que múltiplos pedidos sejam decididos para um única posição do *log*. Vale destacar que ambas as estratégias citadas também poderiam ser aplicadas ao WPaxos, apesar de *batching* não ser tão relevante para a estratégia de sub-divisão do *log* que este utiliza.

Figura 28 – Latência 99 percentil (6 nodos)



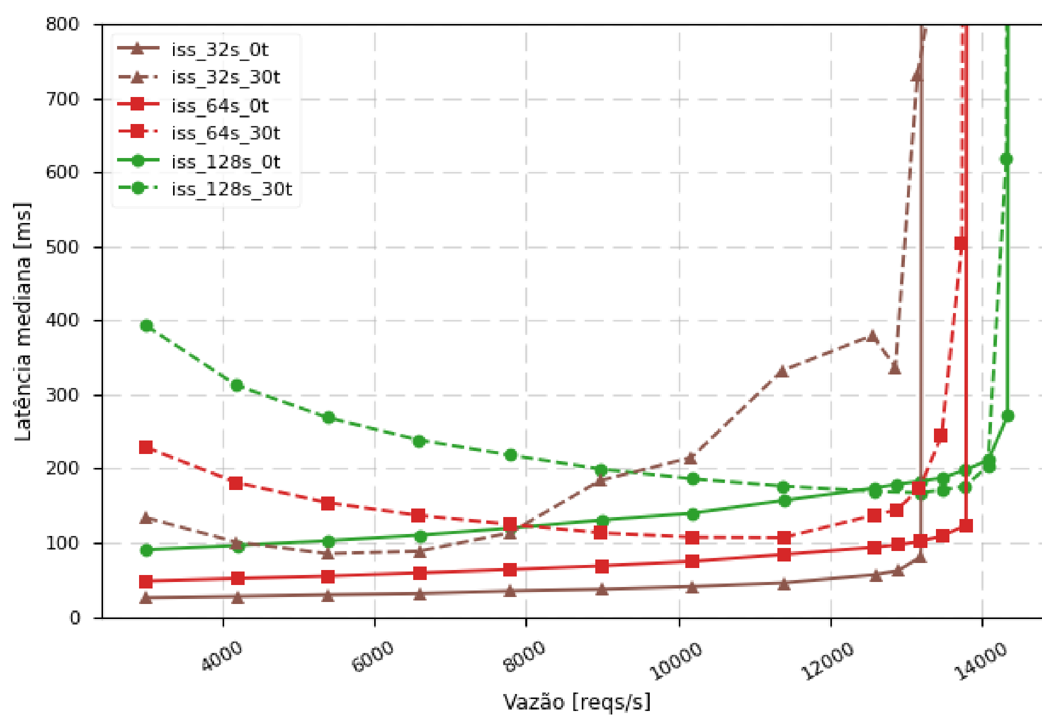
Fonte: De autoria própria.

Figura 29 – Comparação de Vazão (Com e sem otimização)



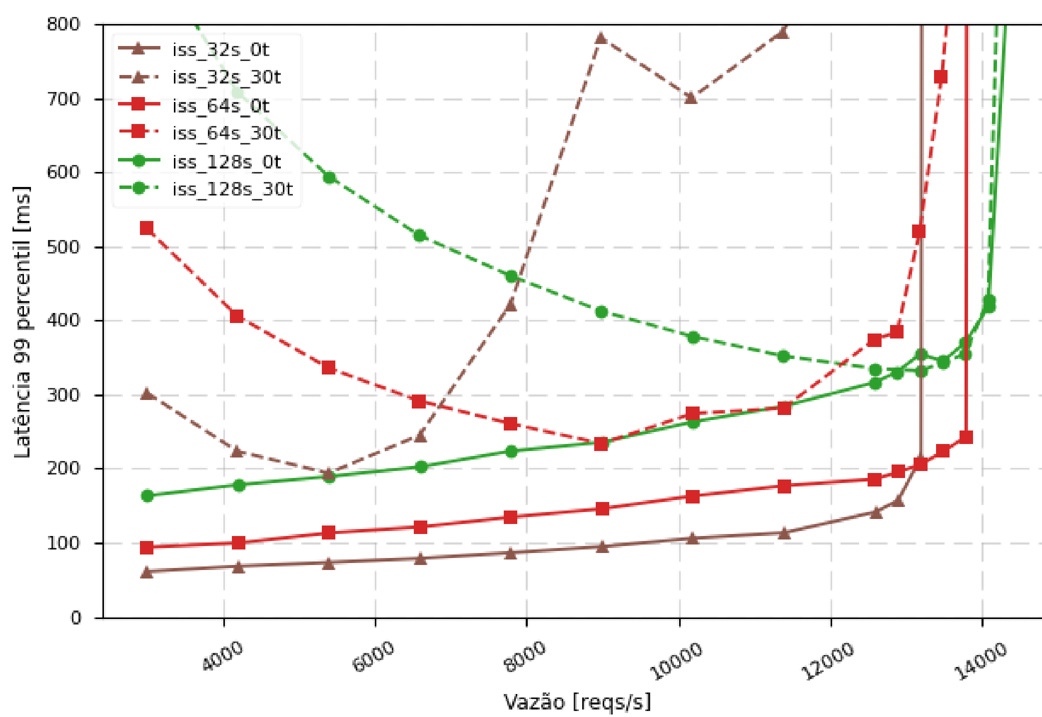
Fonte: De autoria própria.

Figura 30 – Latência Mediana (6 nodos otimizado)



Fonte: De autoria própria.

Figura 31 – Latência 99 percentil (6 nodos otimizado)



Fonte: De autoria própria.

6 CONCLUSÃO

Este trabalho explorou a implementação e aplicação do *framework* ISS ao protocolo Paxos na biblioteca Paxi, avaliando seu desempenho em relação a outros algoritmos de consenso distribuído multi-líderes de alta vazão, o WPaxos e EPaxos. Para isso, foi explorado o tema de algoritmos de consenso distribuídos, caracterizando seus principais usos, características e alguns de seus modos de execução. Dos algoritmos de consenso existentes, foi dado especial foco para o Paxos, com sua execução sendo detalhada e suas otimizações mais comuns discutidas. Além disso, foram apresentadas possíveis ferramentas para a implementação do algoritmo pretendido, com o Paxi sendo escolhido devido a sua modularidade e pelo fato de já possuir outros algoritmos de consenso implementados, que poderiam ser utilizados para comparação com a versão do *framework* ISS desenvolvida.

De forma a melhor entender o problema de escalabilidade dos processos de consenso, foi realizada uma extensiva pesquisa sobre variantes do protocolo Paxos. Foram apresentadas variantes que tem como foco a reconfiguração do algoritmo, já que esta se torna mais comum com o acréscimo de nodos, sendo necessárias para algoritmos escaláveis. Além disso, foram abordadas estratégias de alterações nas estruturas de envio e recebimento de mensagens do protocolo, que possuíssem como objetivo redução de latência do algoritmo, já que estas por vezes podem ser combinadas com outras estratégias de aumento de vazão. Por fim, foram exploradas as variantes multi-líder do protocolo com especial foco para o WPaxos e EPaxos, que foram escolhidos para comparação com o algoritmo implementado por apresentarem duas estratégias diferentes de multiplexação do envio de propostas pelos nodos.

Após este levantamento, foi detalhado o algoritmo do *framework* ISS apresentado em Stathakopoulou, Pavlovic e Vukolić (2022), assim como os detalhes de sua implementação na biblioteca Paxi, que envolveu a criação de uma estrutura de grupo de *buckets* e aplicação de estratégias de controle de concorrência, para se utilizar de paralelismo de co-rotinas durante a execução deste. Além disso, foram apresentados também as adaptações ao protocolo Paxos, de forma que este pudesse ser executado como uma instância de *Sequenced Broadcast* (SB) compatível com o *framework*. Estas adaptações incluem a implementação de um simples detector de falhas, alteração do modelo de proposta do algoritmo, que passa a ser ativo e utilizar mensagens *SKIP*, e a utilização de uma co-rotina trabalhadora para linearizar a execução das funções de envio e recebimento de mensagens. Além destas adaptações, uma otimização baseada no Mencius (MAO; JUNQUEIRA; MARZULLO, 2008) foi implementada, que reduz o número de mensagens necessárias para um valor vazio ser decidido, melhorando o desempenho do algoritmo.

Por fim, foram realizados experimentos no Emulab, de forma a simular a execução do algoritmo desenvolvido em comparação com o WPaxos e EPaxos. A partir destes experimentos, foi possível concluir que o *framework* implementado apresenta alta vazão

em relação a estes algoritmos quando executando 3 ou 6 nodos devido ao seu elevado grau de paralelização da execução do algoritmo de consenso. No entanto, este também apresentou altas latências medianas independentemente de sua configuração. Isto é causada pelo particionamento no espaço de clientes entre os líderes de cada instância de SB, que faz com que determinados pedidos não possam ser propostos até que uma reconfiguração seja executada, adicionando um tempo de espera adicional para que o pedido possa ser decidido e executado. Além disso, o ISS também apresentou uma maior queda percentual de vazão entre as execuções com 3 e 6 nodos quando comparado ao WPaxos, o que pode indicar que este apresentaria piores resultados quando em ambientes com mais nodos, sendo que mais experimentos precisam ser realizados de forma a confirmar ou negar esta hipótese.

6.1 TRABALHOS FUTUROS

Para trabalhos futuros, seria possível avaliar o desempenho do *framework* implementado com uma quantidade maior de nodos, como por exemplo 128, de forma a melhor avaliar sua escalabilidade. Neste caso, outras estratégias de redução de latência, como quorums flexíveis ou redistribuidores conforme apresentado no *Pig* Paxos, também poderiam ser aplicadas sobre a instância de *Sequenced Broadcast* sendo avaliada, já que estas se mostram mais relevantes em cenários com grande número de nodos.

Além destas modificações, o ISS também poderia se beneficiar do uso de *batches*, que potencialmente aumentaria a vazão do protocolo e tornaria as configurações de aguardo por pedidos mais relevantes. Por fim, outra possibilidade seria a realização de experimentos voltados para avaliar o potencial de reconfiguração do *framework*, quando comparado a outras variantes voltadas para este aspecto, por meio de simulações de falhas nos nodos durante sua execução, já que esta não foi tão explorada no artigo original e comparações de recuperação de falhas não são tão comuns na pesquisa de algoritmo multi-líder, já que estes estão mais focados na escalabilidade em si.

REFERÊNCIAS

AILIJIANG, Ailidani; CHARAPKO, Aleksey; DEMIRBAS, Murat. Dissecting the Performance of Strongly-Consistent Replication Protocols. *In:* PROCEEDINGS of the 2019 International Conference on Management of Data. [*S.l.*: *s.n.*], 2019. P. 1696–1710.

AILIJIANG, Ailidani; CHARAPKO, Aleksey; DEMIRBAS, Murat; KOSAR, Tevfik. WPaxos: Wide Area Network Flexible Consensus. **IEEE Trans. Parallel Distrib. Syst.**, IEEE Press, v. 31, n. 1, p. 211–223, jan. 2020. ISSN 1045-9219. DOI: `10.1109/TPDS.2019.2929793`. Disponível em: `https://doi.org/10.1109/TPDS.2019.2929793`.

BENZ, Samuel; BEZERRA, Eduardo. **URingPaxos : A high throughput atomic multicast protocol**. 2013. Disponível em: `https://github.com/sambenz/URingPaxos`. Acesso em: 29 set. 2024.

BESSANI, Alysson; SOUSA, João; ALCHIERI, Eduardo E.P. State Machine Replication for the Masses with BFT-SMART. *In:* 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. [*S.l.*: *s.n.*], 2014. P. 355–362. DOI: `10.1109/DSN.2014.43`.

BURROWS, Mike. The Chubby Lock Service for Loosely-Coupled Distributed Systems. *In:* PROCEEDINGS of the 7th Symposium on Operating Systems Design and Implementation. Seattle, Washington: USENIX Association, 2006. (OSDI ’06), p. 335–350.

CACHIN, Christian; GUERRAOUI, Rachid; RODRIGUES, Luís. **Introduction to Reliable and Secure Distributed Programming**. [*S.l.*]: Springer Berlin Heidelberg, 2011. ISBN 9783642152603. DOI: `10.1007/978-3-642-15260-3`. Disponível em: `http://dx.doi.org/10.1007/978-3-642-15260-3`.

CASTRO, Miguel; LISKOV, Barbara. Practical byzantine fault tolerance and proactive recovery. **ACM Trans. Comput. Syst.**, Association for Computing Machinery, New York, NY, USA, v. 20, n. 4, p. 398–461, nov. 2002. ISSN 0734-2071. DOI: `10.1145/571637.571640`. Disponível em: `https://doi.org/10.1145/571637.571640`.

CHARAPKO, Aleksey. **Pile of Eternal Rejections: Revisiting Mencius SMR**. 2024. Disponível em: `https://charap.co/pile-of-eternal-rejections-revisiting-mencius-smr/`. Acesso em: 19 out. 2024.

CHARAPKO, Aleksey; AILIJIANG, Ailidani; DEMIRBAS, Murat. PigPaxos: Devouring the Communication Bottlenecks in Distributed Consensus. *In:* PROCEEDINGS of the 2021 International Conference on Management of Data. Virtual Event, China: Association for Computing Machinery, 2021. (SIGMOD ’21), p. 235–247. DOI: `10.1145/3448016.3452834`. Disponível em: `https://doi.org/10.1145/3448016.3452834`.

CONWAY, Melvin E. Design of a separable transition-diagram compiler. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 6, n. 7, p. 396–408, jul. 1963. ISSN 0001-0782. DOI: 10.1145/366663.366704. Disponível em: https://doi.org/10.1145/366663.366704.

CORBETT, James C. *et al.* Spanner: Google's Globally-Distributed Database. *In:* PROCEEDINGS of the 10th USENIX Conference on Operating Systems Design and Implementation. Hollywood, CA, USA: USENIX Association, 2012. (OSDI'12), p. 251–264.

CUI, Haochuan *et al.* **phxpaxos**. 2018. Disponível em: https://github.com/Tencent/phxpaxos. Acesso em: 29 set. 2024.

DIJKSTRA, Edsger W. Guarded commands, nondeterminacy and formal derivation of programs. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 18, n. 8, p. 453–457, ago. 1975. ISSN 0001-0782. DOI: 10.1145/360933.360975. Disponível em: https://doi.org/10.1145/360933.360975.

DWORK, Cynthia; LYNCH, Nancy; STOCKMEYER, Larry. Consensus in the presence of partial synchrony. **Journal of the ACM (JACM)**, ACM New York, NY, USA, v. 35, n. 2, p. 288–323, 1988.

HUNT, Patrick *et al.* ZooKeeper: Wait-Free Coordination for Internet-Scale Systems. *In:* PROCEEDINGS of the 2010 USENIX Conference on USENIX Annual Technical Conference. Boston, MA: USENIX Association, 2010. (USENIXATC'10), p. 11.

JUNQUEIRA, Flavio P.; REED, Benjamin C.; SERAFINI, Marco. Zab: High-performance broadcast for primary-backup systems. *In:* PROCEEDINGS of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks. USA: IEEE Computer Society, 2011. (DSN '11), p. 245–256. DOI: 10.1109/DSN.2011.5958223. Disponível em: https://doi.org/10.1109/DSN.2011.5958223.

LAMPORT, L.; MASSA, M. Cheap Paxos. *In:* INTERNATIONAL Conference on Dependable Systems and Networks, 2004. [*S.l.*: *s.n.*], 2004. P. 307–314. DOI: 10.1109/DSN.2004.1311900.

LAMPORT, Leslie. Fast Paxos. **Distrib. Comput.**, Springer-Verlag, Berlin, Heidelberg, v. 19, n. 2, p. 79–103, out. 2006. ISSN 0178-2770. DOI: 10.1007/s00446-006-0005-x. Disponível em: https://doi.org/10.1007/s00446-006-0005-x.

________. Lower Bounds on Consensus. [*S.l.*], mar. 2000. Disponível em: https://www.microsoft.com/en-us/research/publication/lower-bounds-consensus/.

________. Paxos Made Simple. **ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)**, p. 51–58, dez. 2001.

Disponível em:
https://www.microsoft.com/en-us/research/publication/paxos-made-simple/.

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

LAMPORT, Leslie; MALKHI, Dahlia; ZHOU, Lidong. Vertical paxos and primary-backup replication. *In:* PROCEEDINGS of the 28th ACM Symposium on Principles of Distributed Computing. Calgary, AB, Canada: Association for Computing Machinery, 2009. (PODC '09), p. 312–313. DOI: 10.1145/1582716.1582783. Disponível em: https://doi.org/10.1145/1582716.1582783.

________.________. *In:* PROCEEDINGS of the 28th ACM Symposium on Principles of Distributed Computing. Calgary, AB, Canada: Association for Computing Machinery, 2009. (PODC '09), p. 312–313. DOI: 10.1145/1582716.1582783. Disponível em: https://doi.org/10.1145/1582716.1582783.

MAO, Yanhua; JUNQUEIRA, Flavio P.; MARZULLO, Keith. Mencius: building efficient replicated state machines for WANs. *In:* PROCEEDINGS of the 8th USENIX Conference on Operating Systems Design and Implementation. San Diego, California: USENIX Association, 2008. (OSDI'08), p. 369–384.

MARANDI, Parisa Jalili; PRIMI, Marco; PEDONE, Fernando. Multi-Ring Paxos. *In:* IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012). [*S.l.*: *s.n.*], 2012. P. 1–12. DOI: 10.1109/DSN.2012.6263916.

MARANDI, Parisa Jalili; PRIMI, Marco; SCHIPER, Nicolas *et al.* Ring Paxos: A high-throughput atomic broadcast protocol. *In:* 2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN). [*S.l.*: *s.n.*], 2010. P. 527–536. DOI: 10.1109/DSN.2010.5544272.

MORARU, Iulian; ANDERSEN, David G.; KAMINSKY, Michael. There is more consensus in Egalitarian parliaments. *In:* PROCEEDINGS of the Twenty-Fourth ACM Symposium on Operating Systems Principles. Farminton, Pennsylvania: Association for Computing Machinery, 2013. (SOSP '13), p. 358–372. DOI: 10.1145/2517349.2517350. Disponível em: https://doi.org/10.1145/2517349.2517350.

OKI, Brian M.; LISKOV, Barbara H. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. *In:* PROCEEDINGS of the Seventh Annual ACM Symposium on Principles of Distributed Computing. Toronto, Ontario, Canada: Association for Computing Machinery, 1988. (PODC '88), p. 8–17. DOI: 10.1145/62546.62549. Disponível em: https://doi.org/10.1145/62546.62549.

ONGARO, Diego; OUSTERHOUT, John. In search of an understandable consensus algorithm. *In:* PROCEEDINGS of the 2014 USENIX Conference on USENIX Annual Technical Conference. Philadelphia, PA: USENIX Association, 2014. (USENIX ATC'14), p. 305–320.

PRIMI, Marco; SCIASCIA, Daniele. **LibPaxos: Open-source Paxos**. 2013. Disponível em: `https://libpaxos.sourceforge.net/`. Acesso em: 22 jun. 2024.

REGIS, Sofia; MENDIZABAL, Odorico. Análise comparativa do algoritmo Paxos e suas variações. *In:* ANAIS do XXIII Workshop de Testes e Tolerância a Falhas. Fortaleza: SBC, 2022. P. 71–84. DOI: `10.5753/wtf.2022.223461`. Disponível em: `https://sol.sbc.org.br/index.php/wtf/article/view/21506`.

STATHAKOPOULOU, Chrysoula; PAVLOVIC, Matej; VUKOLIĆ, Marko. State machine replication scalability made simple. *In:* PROCEEDINGS of the Seventeenth European Conference on Computer Systems. Rennes, France: Association for Computing Machinery, 2022. (EuroSys '22), p. 17–33. DOI: `10.1145/3492321.3519579`. Disponível em: `https://doi.org/10.1145/3492321.3519579`.

VAN RENESSE, Robbert; ALTINBUKEN, Deniz. Paxos Made Moderately Complex. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 47, n. 3, fev. 2015. ISSN 0360-0300. DOI: `10.1145/2673577`. Disponível em: `https://doi.org/10.1145/2673577`.

VASSANTLAL, Robin *et al.* **Byzantine Fault-Tolerant (BFT) State Machine Replication (SMaRt)**. 2024. Disponível em: `https://github.com/bft-smart/library`. Acesso em: 29 set. 2024.

VERÍSSIMO, Paulo; RODRIGUES, Luís. **Distributed Systems for System Architects**. [*S.l.*]: Springer US, 2001. ISBN 9781461516637. DOI: `10.1007/978-1-4615-1663-7`. Disponível em: `http://dx.doi.org/10.1007/978-1-4615-1663-7`.

WHITE, Brian *et al.* An Integrated Experimental Environment for Distributed Systems and Networks. *In:* USENIX ASSOCIATION. PROC. OF the Fifth Symposium on Operating Systems Design and Implementation. Boston, MA: [*s.n.*], dez. 2002. P. 255–270.

Apêndices

APÊNDICE A – IMPLEMENTAÇÃO DO ISS NA BIBLIOTECA PAXI

O código fonte pode ser encontrado em: https://github.com/g-buzzi/iss-paxos. —

Anexos

ANEXO A – ARTIGO DO PROJETO

Application of the ISS framework to Paxos Papers and Abstracts

Gabriel Momm Buzzi[1], **Odorico M. Mendizabal**[1]

[1]*Departamento de Informatica e Estaátisica, Universidade Federal de Santa Catarina UFSC*
Florianópolis, Santa Catarina, Brazil

gabriel.m.buzzi@grad.ufsc.br, odorico.mendizabal@ufsc.br

Abstract. *This paper applies the ISS framework, capable of increasing scalability and throughput of consensus protocols by utilizing a multi-leader strategy, to the Paxos protocol, evaluating its performance pertaining to throughput, latency and scalability when compared to other multi-leader protocols, such as WPaxos and EPaxos. The ISS framework was implemented in Go by utilizing the Paxi library. Experimental results showed that the implemented framework presented a better throughput than other multi-leader protocols that were evaluated, but also presented an elevated latency and potentially worse scalability.*

1. Introduction

Distributed consensus protocols are those that allow multiple process to decide on a single value based on a set of proposed values. These algorithms are essential to distributed services such as leader election, reliable broadcast and replication, by allowing them to safely decide a value even in the eventuality of process failure, by guaranteeing the termination, validity, integrity and agreement of all decisions made [Cachin et al. 2014].

One of the main challenges in the development of consensus protocols is its scalability in relation to the number of participating nodes [Stathakopoulou et al. 2022, Ailijiang et al. 2020, Moraru et al. 2013]. In order to assure the integrity of the protocol, with the increase in the number of nodes, more messages need to be sent and processed for each decision made, leading to reduced throughput and increased latency. Adding to this limitation, many protocols, such as Paxos [Lamport 1998], Raft [Ongaro and Ousterhout 2014] and PBFT [Castro and Liskov 1999], also operate with a single leader process. The leader is the only one capable of proposing values and is usually assigned a higher proportion of the processing load of messages, leading it to becoming a bottleneck in the algorithm performance.

Many algorithms have tried to solve this bottleneck problem, be it by changing how messages are sent [Jalili et al. 2010, Charapko et al. 2021], by trading availability for better performance [Howard et al. 2016] or by allowing multiples simultaneous leaders [Mao et al. 2008, Ailijiang et al. 2020, Moraru et al. 2013]. Multi-leader protocols are one of the most prominent strategies to increasing the scalability of a protocol. With more leaders, the load of sending and processing messages can be better distributed, eliminating the bottleneck problem and increasing throughput without sacrificing the resilience of the algorithm.

The Insanely Scalable SMR (ISS) [Stathakopoulou et al. 2022] is a framework capable of making single leader consensus protocols scale with the number of nodes

by safely parallelizing their execution, effectively turning them into multi-leader ones. This paper evaluates the performance of a version of this framework applied to the Paxos consensus protocol with the objective of quantifying its throughput, latency and scalability when compared to other multi-leader protocols. For this, a version of the framework and a variant of Paxos were implemented in Go [1] with the utilization of the Paxi [Ailijiang et al. 2019] library.

2. Related Work

This section gives an overview of the Paxos Consensus protocol and some of its variants.

2.1. Paxos Protocol

Paxos [Lamport 1998] is one of the first consensus protocols created, being developed by Leslie Lamport in 1998. During its execution, each participating process takes at least one of the following roles:

- **Proposer:** Proposes values, such that one is eventually decided. A ballot is utilized to indicate the process that made the proposal and how recent it is.
- **Acceptor:** Stores the most recently accepted proposals, serving as the memory of the algorithm.
- **Learner:** Responsible for learning what value has been decided.

These process exchange messages in two phases of communication, that can be repeated across multiple rounds, as can be seen in Figure 1. During Phase 1, we have the leader election, in which a proposer sends a message P1A, with a ballot number, to all acceptors, asking them to not accept ballot numbers lower than the one sent. An acceptor, receiving this message, compares the received ballot number to the highest ballot number known. If the received value is lower, the message is ignored. Otherwise, it sends a P2B response to the proposer, containing: the accepted ballot number, the last value accepted and the ballot in which that value was accepted.

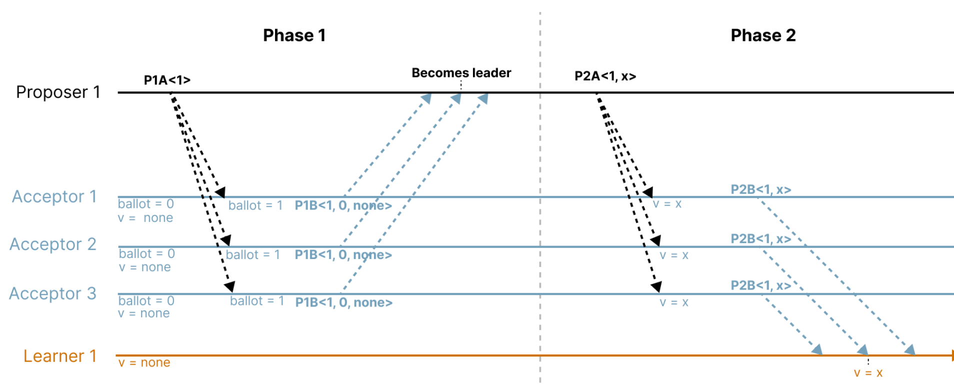


Figure 1. Normal execution of the Paxos protocol

Upon receiving a majority of responses, the proposer is considered the leader of the algorithm and starts phase 2 by sending, to all acceptors, a P2A message containing: its ballot number and a value that it wants accepted. The acceptors, upon receiving this message, compare the received ballot number with the highest known one, discarding

[1] https://github.com/g-buzzi/iss-paxos

the message in cases where the received ballot is lower. Otherwise, it accepts the value, sending to all learners a P2B message that contains the ballot number and the accepted value. A learner may then consider a value to be decided when it receives a majority of P2B messages for a single ballot containing the same value.

This decided value is guaranteed to not change in the future due to the way a leader selects a value after executing phase 1 of the algorithm. As can be seen in Figure 2, when a value has already been accepted, the P1B response contains said value and the ballot under which it was accepted. With this information, the newly elected leader can always choose to either: propose the value associated to the highest ballot number or, in case no value has been accepted yet, propose a new value. As messages from older ballots are ignored and quorums from different phases always intercept (since they are a majority) Paxos guarantees that, once a certain value v is decided, it can never change.

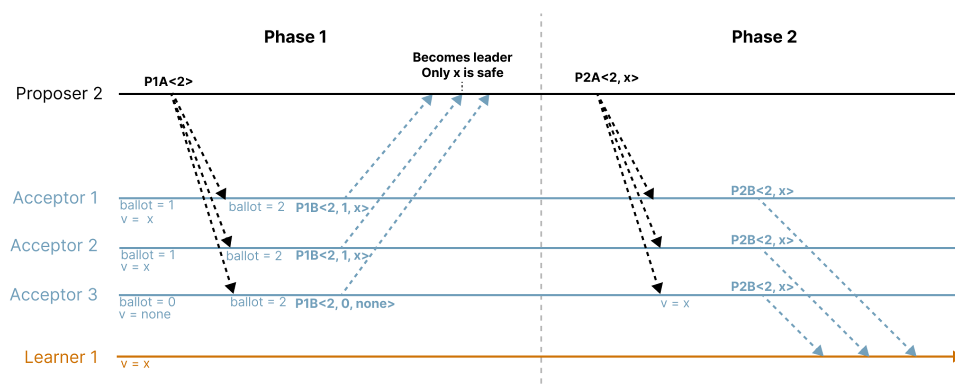


Figure 2. Example of a Paxos execution where a value is already decided

In practical cases, in order to reduce the number of messages sent, some optimizations are usually made to the Paxos protocol. Most commonly, all nodes execute all three roles (proposer, acceptor and learner) and, in order to reduce the number of messages P2B sent, the leader node also acts as a distinguished learner. The distinguished learner is the only one that will receive the P2B responses, that are no longer broadcast to all acceptors. Once a value has been decided, this learner then sends a commit message (P3) to all other learners, containing the decided value and indicating that this is the value that was decided.

The original Paxos is also limited in that it can only decide a single value. As many applications, such as reliable broadcast and replication, need to decide a theoretically infinite number of values, a common extension of Paxos is **Multi-Paxos** [Lamport 2001]. In Multi-Paxos, a sequence of n values are decided by creating n instances of the Paxos algorithm and utilizing each of these to decide one value in the sequence. A common optimization for Multi-Paxos is to make it so phase 1 of the algorithm is executed simultaneously in all instances of Paxos, making it so that a single leader exists. With this optimization, as long as this leader does not fail, phase 1 can be executed only once for all decisions made, reducing the number of messages exchanged. Having a single leader also allows to safely and simultaneously propose multiple values across instances of Multi-Paxos.

2.2. Paxos Variants

Taking Paxos and Multi-Paxos as a base, multiple variants have been created throughout the years, each focused on improving a different aspect of the algorithm, such as introducing the possibility of reconfiguration, reducing latency or increasing its scalability.

2.2.1. Reconfiguration

The need for reconfiguration in consensus protocols comes from the fact that, in a practical environment, the list of participating process does not tend to remain static, with faulty nodes being removed and new nodes being added in order to increase fault tolerance. As such, many strategies to allow reconfiguration [Lamport and Massa 2004, Lamport et al. 2009] have been proposed.

One of these strategies, is that of **Cheap Paxos** [Lamport and Massa 2004], which utilizes a special class of reconfiguration commands that can add or remove nodes from the list of participating processes. These commands can be proposed by the protocol leader, in the same way as other values. Upon being decided, the command is then executed and the list of participating processes is updated. This strategy is a simple way of implementing reconfiguration, but it also limits the number of values that can be simultaneously proposed by making it so that, until the value of instance i is decided and executed, the quorum for instance $i + 1$ can not be precisely determined, making it impossible to decide a value in it.

Another approach for reconfiguration can be seen in **Vertical Paxos** [Lamport et al. 2009], in which a reconfiguration is decided upon by a configuration master from outside of the consensus protocol. In this variant, when a node is suspected of failure or the system can no longer progress, one of its participants may send a message to the reconfiguration master, that responds with: a new configuration, a new leader protocol and a new ballot. With this information, the new leader executes phase 1 of the consensus protocol, attempting to effectively become the leader. With this, in Vertical Paxos, since each configuration is attached to a single ballot, it is possible to simultaneously propose multiple values in different instances, since the change in configuration is accompanied by a execution of a leader election that guarantees safety.

2.2.2. Message Exchange

One of the main factors that influences the latency and scalability of a consensus protocol, is the way in which messages are sent and received by each process. This is because the number of rounds of communication necessary for a decision directly affects the latency of the protocol, with more rounds usually leading to a longer time before a value is decided. Due to hardware limitations, another important characteristic is the total number of messages being sent per decision, since with fewer messages being sent per decision, more decisions can be made with the same amount of messages. How the processing load of these messages is distributed among processes is also an important factor, since unbalanced loads can lead to the creation of an execution bottleneck.

In Multi-Paxos, due to the fact that a single leader exists for all instances, an ad-

ditional round of communication is usually created in the form of a forwarding message. As can be seen in Figure 3 (a), when a node that is not the current leader receives a request (1), this must first be forwarded to the current leader (2) in order for each to be proposed with a P2A message (3). This intermediate step increases the latency of each request not directly sent to the leader, reducing the perceived speed of the protocol. **Fast Paxos** [Lamport 2006] is a variant that removes this additional forwarding step by introducing fast rounds.

A fast round is one in which the leader can, instead of proposing a value, send a especial P2A "Any" message, that indicates to the acceptors that they may accept any value sent by other proposers as if it was sent by the leader. As shown in Figure 3 (b), during a fast round the proposer sends its proposal directly to the acceptors (1), that then send their responses to the protocol leader (2). The leader, based on these responses, determines if a value was decided. In case a quorum was formed, the leader can consider the value decided. Otherwise, if multiple processes simultaneously proposed values and none was chosen, the leader them starts a new slow round, that is executed similarly to a normal Paxos round, but skips the execution of phase 1 by replacing the role of P1B messages with the information received in the P2B responses already received. Lastly, it's important to note that, in order to guarantee integrity, quorums in a fast round have increased restraints, that makes it so either 3 quarters of the acceptors are required for fast quorums or that all quorums have an increased size 2 thirds of all acceptors.

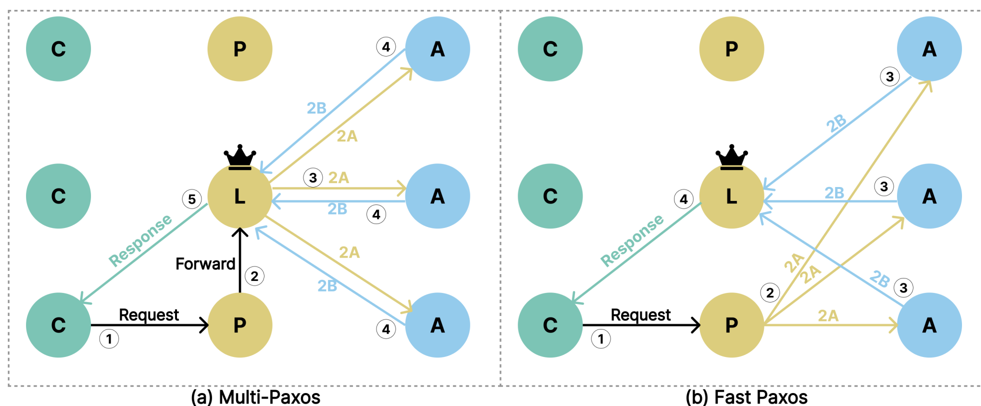


Figure 3. Example of phase 2 execution for Multi-Paxos and Fast Paxos

Another strategy to increasing scalability and the number of messages sent per decision, is to change the way quorums are formed. In the original Paxos protocol, quorums, in any phase of the algorithm, are always formed by a majority of acceptors, requiring, in a system with N nodes, at least $(N/2) + 1$ messages to reach a decision. **Flexible Paxos** [Howard et al. 2016] is a variant that acknowledges that this definition is more conservative than necessary. Instead of a majority, the only requirement for quorums to be formed, in a way that guarantees consistency, is to make it so that all quorums for phase 1 intersect all quorums of phase 2.

By utilizing this definition, it's possible to create more flexible quorums in which, for example, we increase the number of nodes required in a phase 1 quorum (Q1), that is rarely used, in order to decrease the nodes necessary to form a phase 2 quorum (Q2),

that is executed for every value decided. Besides this trade-off, it's also possible to create specific quorums structures that guarantee this interception, such as creating a grid of acceptors, such that its rows are quorums for phase 1 and its columns are quorums for phase 2. As seen in Figure 4, this allows for the creation of even smaller phase 2 quorums, at the cost of making it so the protocol tolerates a smaller number of faults and that the fault of certain nodes may be more or less detrimental to the system.

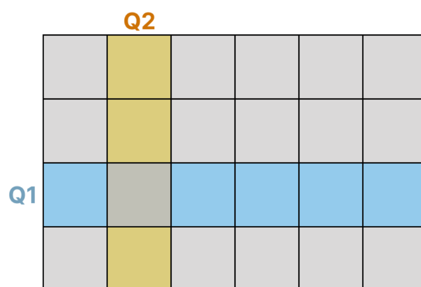


Figure 4. Example of a grid quorum

Lastly, it is also possible to improve the throughput and scalability of a protocol by better distributing the processing load of messages across all nodes. In the original Paxos algorithm, most nodes only send and receive a constant amount of messages per decision made, being responsible only for receiving a proposal and sending a response. The exception to this behavior is the protocol leader, that must process a number of messages that scales with the number of participating nodes, such that the leader commonly becomes a hardware bottleneck to the increase of throughput [Ailijiang et al. 2020, Moraru et al. 2013, Mao et al. 2008, Stathakopoulou et al. 2022]. To distribute the processing load, it's possible to use strategies such as those presented in Ring Paxos [Jalili et al. 2010] and Pig Paxos [Charapko et al. 2021].

In **Ring Paxos** [Jalili et al. 2010], the proposer during phase 1, besides sending a ballot, also sends a logical ring configuration, in the form of a sequence of nodes, that includes a quorum of acceptors and ends on the leader process. This configuration is used during P1B and P2B responses, as seen in Figure 5 (a), by making it so that the first acceptor in the ring, instead of directly sending its response to a leader, forwards it to the next node in the ring configuration. This node then combines the response with its own and forwards it to the next leader, with this step being repeated until al the responses are forwarded to the leader, that learn if the value was decided.

Another strategy is that of **Pig Paxos** [Charapko et al. 2021], that utilizes a hierarchical configuration, in which the acceptors acceptors are divided in relay groups, as seen in Figure 5 (b). When a proposer must send a P1A or P2A message, instead of broadcasting it to all acceptors, sends it to one randomly selected member of each relay group, that acts as a relay node. A relay node is responsible for forwarding the message to all other acceptors in its relay group, receiving their responses and, in a single message, forwarding all responses to the leader. This method reduces the quantity of messages processed by the leader, but also increases the communication rounds necessary for a decision, increasing response latency, and also introducing potential failure at the relay group level.

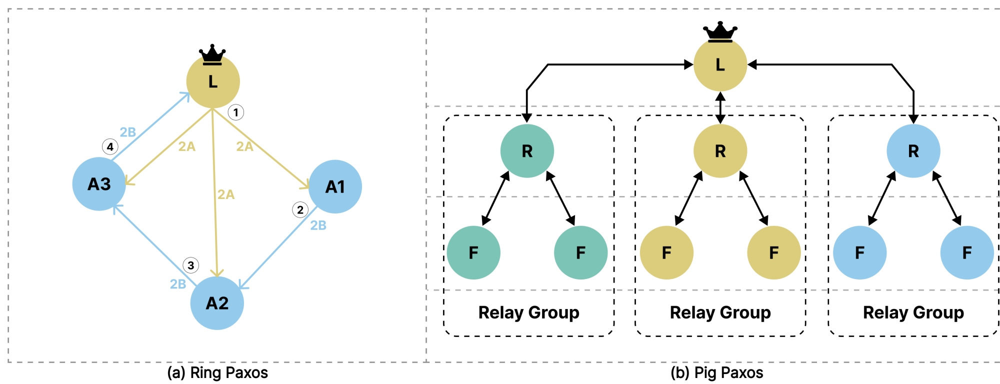


Figure 5. Messaging structure of Ring Paxos and Pig Paxos

2.2.3. Multi-Leader Protocols

Multi-Leader protocols are those that allows multiple processes to propose values simultaneously, by making it so all of them act as leaders or by not requiring a leader at all. This strategy increases protocol scalability and solves the leader bottleneck problem by making it so the addition of new nodes increases not only the number of messages processed, but also the number of nodes available for processing them.

One of the most intuitive strategies for the creation of multi-leader protocols is that of **Mencius** [Mao et al. 2008], in which the sequence of values to be decided is partitioned, round-robin fashion, allowing for a different protocol to be leader of each partition. However, as different leaders may have different speeds, this strategy can create gaps in the log, such as seen in Figure 6, in which only command x_0 may be executed even though other commands have been decided, since commands must be executed in order and the second partition still hasn't decided on a value for position 1. To avoid these gaps, a leader in Mencius will send a Skip message, that indicates a instance will have no value decided, when it perceives it's behind other leaders. These Skip messages are also optimized by being decided in a single P2A message, something that doesn't violate integrity due to Mencius only allowing the initial leader of each partition to propose a value different from SKIP.

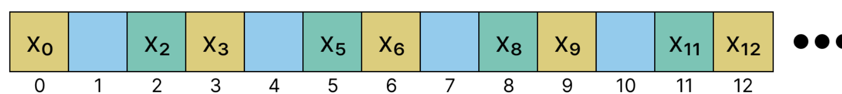


Figure 6. Example of a Mencius of with gaps (different colors represent different partitions)

Another strategy is that of Wan Paxos (WPaxos) [Ailijiang et al. 2020], a variant focused on reducing latency and increasing scalability for Key-Value databases in Wide-Area Network (WAN), by parallelizing the protocol execution. To achieve this, WPaxos creates a separate instance of Multi-Paxos for each key present in the Key-Value store, allowing for each one to have a separate leader and for decisions to be completely independent. Besides this, WPaxos also optimizes the latency of WAN requests by utilizing

flexible grid quorums, to form phase 2 quorums with geographically close nodes, and by utilizing an adaptive steal strategy for phase 1, that tries to make it so the leader of each key in the system is closer to areas where access to those keys is more likely. Although very scalable, WPaxos has the disadvantage of only being applicable in systems that can fully partition that values that need to be decided, not requiring operations that affect values from different partitions at once.

Lastly, Egalitatian Paxos (EPaxos) [Moraru et al. 2013] is a variant that, instead of directly deciding on a ordered sequence of values, decides a set of dependencies and a sequence number for each value. These dependencies indicate which commands have already been accepted and that interfere with the execution of the new command. Based on these dependencies and the sequence numbers, a dependency graph can be created to determine the partial order in which commands must be executed, in such a way that the execution is realizable in all nodes. With this, EPaxos is a leaderless protocols, where all nodes are equal to each other, and may decide values through a Fast Path, similarly to Fast Paxos [Lamport 2006], where if all dependencies returned by a quorum of acceptors is the same, those dependencies are considered decided. Otherwise, in case multiple conflicting proposals have reached acceptors in different orders and an consensus is not reached, the value is decided through additional rounds of messages, in what is called the Slow Path. This makes it so the performance of this algorithm is directly tied to how many conflicting commands are proposed simultaneously, with more conflicting commands leading to more commands being decided in the Slow Path and more messages needed to decide a value on average.

3. ISS

Insanely Scalable SMR (ISS) [Stathakopoulou et al. 2022] is a framework capable of increasing the throughput and scalability of a single leader consensus protocols by parallelizing its executing, making it behave like a multi-leader one. For this, ISS utilizes instances of consensus protocols that implements Sequenced Broadcast (SB), that decides a pre-determined number n of commands before stopping. Besides this, for each instance there exists an initial leader, which is the only process capable of proposing non-empty values. Just as in Mencius, the sequence of values to be decided, called log, is partitioned in s non-intercepting segments, by utilizing a simple round-robin algorithm.

With each partition being assigned an instance of Sequenced Broadcast with its own separate leader, commands can be decided in parallel without interference, but this approach also brings new consistency concerns. With this implementation, it would be possible for client's commands to be decided out of order if they are sent to different instances, as can be seen in Figure 7, where, after commands $C1.1$ and $C1.2$ where decided by instance 0, by sending command $C1.3$ to a different instance, it can be decided before command $C1.2$. To avoid this and other problems, ISS utilizes buckets. A bucket is a set of commands that contains all the commands requested by a subset clients. By assigning each bucket to a single segment, such that only the leader of that segment is capable of proposing values present in it, it's then possible to avoid any potential problem related to the order of decided client requests.

This strategy, however, makes it so that, in case of leader failure, the system will appear unavailable to all clients which requests are mapped to that particular leader's

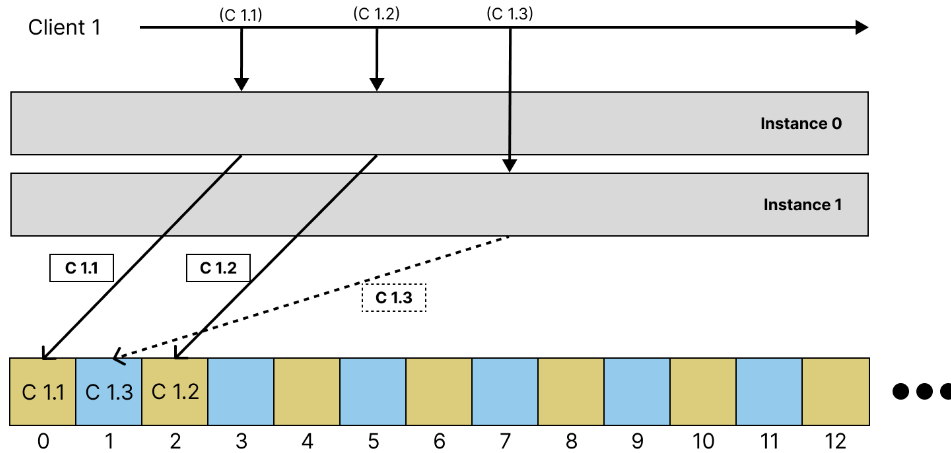


Figure 7. Example of a inconsistent order for client requests

buckets, since only the initial leader may propose non-empty values. To avoid this, ISS utilizes the concept of epochs. Each epoch is a contiguous partition of the log of size e, composed of multiple segments, as seen in Figure 8, in which each segment is represented by a different color. By sequentially executing these epochs, i.e. only executing epoch $i + 1$ after all values of epoch i have been decided, we can utilize the period of transition between epochs, in which all nodes know all commands decided so far, to realize determinist reconfiguration operations, such as rotating the buckets between leaders, to avoid unavailability due to leader failure, or execute reconfiguration operations based on a failure detector or reconfiguration commands.

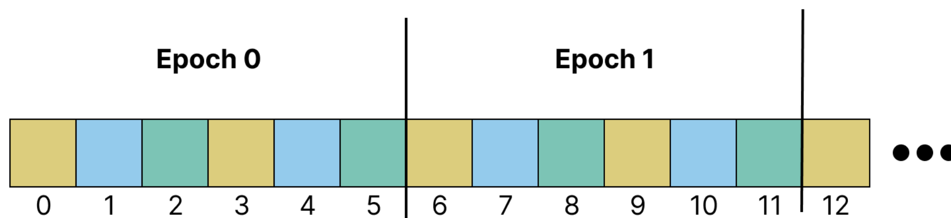


Figure 8. Example of ISS' epoch structure (each color represents a different segment)

3.1. Implementation

The ISS framework described above was implemented in Go [2], with the utilization of the Paxi library [Ailijiang et al. 2019] [3]. The library was responsible for implementing communication between nodes, a key-value data-store (used as an application being replicated by the protocol), the quorums for phase 1 and 2 and a workload generator, later used for evaluation. The implementation of the ISS framework itself consisted on the creation of auxiliary classes, such as buckets and bucket groups, the framework itself,

[2] `https://github.com/g-buzzi/iss-paxos`
[3] `https://github.com/ailidani/paxi`

implemented by a coordinator class, and an adapted version of Paxos, that implements sequenced broadcast.

3.1.1. Auxiliary Classes

For the **auxiliary classes**, the buckets where implemented as a doubly linked lists that stores a client request and a boolean value that indicates wether if this was decided. Besides this, to improve performance, a key-value dictionary is utilized to map client commands to these items, making them easier to locate. Finally, in order to avoid simultaneous access to a single bucket, a mutex was utilized.

In order to allow each segment to easily obtain a request, a bucket group class was also implemented. This bucket groups has a list of all the buckets assigned to a specific segment and is responsible for implementing a **getRequest** method, that retrieves a command from one of these buckets. This **getRequest** method, on not finding any values to be proposed, may be set to immediately return an empty value, indicating a no-op command should be proposed. Additionally, it may also be configured to wait a predetermined time, during which, in case a new request is added to one of its bucket, the request is returned. Otherwise, after the allotted time has passed, an empty value is returned.

3.1.2. Coordinator

The **coordinator** implements the ISS protocol itself, being responsible for: managing the epochs, from its creation until its end; processing requests; forwarding messages to the correct segments; updating the log when prompted by a segment and delivering commands when possible. In order to effectively and safely execute these functions, 4 co-routines, each responsible for a different aspect of the algorithm, are executed in parallel: **handleRequest**, **runEpoch**, **bufferManager** and **logManager**. The **handleRequest** co-routine is responsible for handling client requests by utilizing a hash function to add them to its correct *bucket*.

The co-routine **runEpoch** is tasked with initializing new epochs and forwarding messages to the correct segments. These functions were implemented in the same co-routine in order to guarantee, without the use of a mutex, that messages are never delivered to a segment of a wrong epoch. This is guaranteed by the fact that only this co-routine may advance the epoch. While initializing a new epoch, the leaders of each segment is selected, each bucket is assigned to a segment and all the instances of SB are created and initialized. Epochs are advanced when a signal from the **logManager** is received, indicating that all values in the current epoch have been decided.

In order to correctly forward messages from to its segments, the **runEpoch** co-routine verifies the epoch and segment to which it message is destined. If the message is destined to an instance of SB from a past epoch, it's ignored. In case the message is destined to a current instance of SB, it's forwarded. Lastly, since messages from future epochs can't be immediately delivered to the correct segment, since it hasn't been created yet, they are sent to the **bufferManager**. This **bufferManager** is responsible for

storing all future messages in a stack and forwarding those to $\mathbf{runEpoch}$ when a new epoch is initialized.

Finally, the $\mathbf{logManager}$ is the one responsible for updating the log when a signal is received from one the SB instances. After updating the log, the position $l + 1$, where l is the last delivered command, is verified such that, if it has already been decided, its value is delivered, l is updated to $l+1$ and the next position is verified until a value can no longer be delivered. Upon delivering a value, if the value is the last one belonging to the current epoch, a signal is sent to $\mathbf{runEpoch}$ in order to end the current epoch and begin the next one.

3.1.3. Adaptations to Paxos

In order to execute Paxos as a Sequenced Broadcast instance, some adaptations were necessary. Firstly, they were limited to deciding an s number of requests, based on the segment's size, with only the initial leader of each SB being capable of proposing a non-empty value. In order to avoid gaps in the log, these leaders had their behavior changed so that, instead of passively waiting for clients requests, a co-routine is created in order to always actively try to propose values by calling the bucket group's $\mathbf{getRequest}$ method. Since methods for sending and receiving messages could not be called at the same time due to concurrency problems, a worker co-routine was also created, in order to parallelize the order of execution of all methods.

In order to guarantee termination, a failure detector was implemented as a simple timer that is restarted each time the protocol progresses towards deciding a value. If the timer is over before being restarted, then the node will attempt to execute phase 1 of the Paxos algorithm in order to become the leader and decide all currently undecided values. In case a value was already proposed for a certain instance, that value will be proposed, otherwise, a SKIP message is sent. As in Mencius [Mao et al. 2008], SKIP messages were later optimized such that only the P2A message is required for it to be decided, since, once a SKIP message has been sent, it indicates that either:

- The initial leader sent it, and since it is the only process capable of proposing a value different from SKIP, no value different from SKIP can be decided.
- A different proposer sent it after becoming a leader, in which case, there's no longer a initial leader capable of proposing values other than SKIP.

4. Evaluation

With the aim of evaluating the throughput, latency and scalability of the implemented algorithm, experiments were made in order to simulate its performance by replicating the client-server model inherent to the Paxi library. For the purpose of evaluating the scalability of the protocol, the experiments were realized with with 3 and 6 nodes. For the sake of comparison, each experiment was also conducted on the implementations of WPaxos and EPaxos already implemented in Paxi. Lastly, unless otherwise stated, the ISS implementation does not use the optimization related to SKIP messages.

4.1. Configuration

The experiments were made in Emulab [White et al. 2002], an on-line platform that offers a variety of physical and virtual machines on-demand for research in distributed systems

and networks. For the experiments, physical nodes of type d430 were used, of which configurations can be seen in Table 1. These nodes were organized in a simulated Local Area Network (LAN), being able to communicate with each other.

Table 1. Configuration of each node

Component	Specification
Processor	Intel Xeon E5-2630 v3
Number of Cores	8
Threads per core	2
RAM	65GB
Network card	1Gbps
Operational System	Ubuntu 20.04 64 bits
Go version	1.22.1

To avoid varying results due to resource sharing, each machine exclusively executed the role of a single node, either client or server. In all experiments, 6 nodes were acting as clients, by utilizing paxi's workload generator. The configuration for each client node can be seen in Table 2. Each workload was generated in a period of 60 seconds, during which a 12000 keys data-store was simulated by each server. Each client node could have 256 concurrent requests, of which all were read operations, in order to avoid collisions in the EPaxos algorithm. To simulate a geographical access pattern, each client utilized a normal distribution, defined by σ and μ, to determine the probability of a key being accessed. The normal distributions were made by utilizing a fixed σ value and varying the value of μ for each client node, making it so that a different subset of keys was more likely to be accessed by each client.

Table 2. Load generator parameters

Parameters	Value
Time (T)	60 segundos
Number of Keys (K)	12000
Write percentage (W)	0%
Concurrency	256
Request distribution	normal
Distribution mean (μ)	$(n \cdot 2000) - 1000$
Distribution standard deviation (σ)	1000

Lastly, to better understand the impact of certain parameters in the execution of the ISS framework, the size of each segment and the time a leader would wait for a client request before proposing a no-op value can be seen in Table 3. It's expected that bigger segment sizes would lead to a higher throughput, due to an increase in possible concurrent proposals and a reduction on the impact of the cost initializing epochs, since less epochs are required to decide the same amount of values. In relation to awaiting for messages, it was supposed that this strategy would reduce the amount of SKIP messages, possibly reducing latency due to wasted network usage. Besides this, unless otherwise stated, SKIP messages were not being utilized with the optimization presented at the end of section 3.

Table 3. ISS parameters

Name	Segment Size	Request waiting time (ms)
ISS_32s_0t	32	0
ISS_32s_30t	32	30
ISS_64s_0t	64	0
ISS_64s_30t	64	30
ISS_128s_0t	128	0
ISS_128s_30t	128	30

4.2. Results

4.3. 3 nodes

For 3 nodes, we can see in Figure 9 that the throughput of the implement framework was almost double that of WPaxos and EPaxos. The higher throughput probably comes from the parallelism present in the ISS framework, that, besides allowing multiple nodes to simultaneously propose values, also better utilizes system resources such as CPU cores and threads, by parallelizing the execution of the algorithm in each node as well. On the effects of the ISS parameters in itself, it's possible to observe a slight increase in throughput when executed with larger segments. As about the waiting time, it does not seem to have much effect in the maximum throughput, since in such situations, it would be rare to find an empty bucket in any of the nodes.

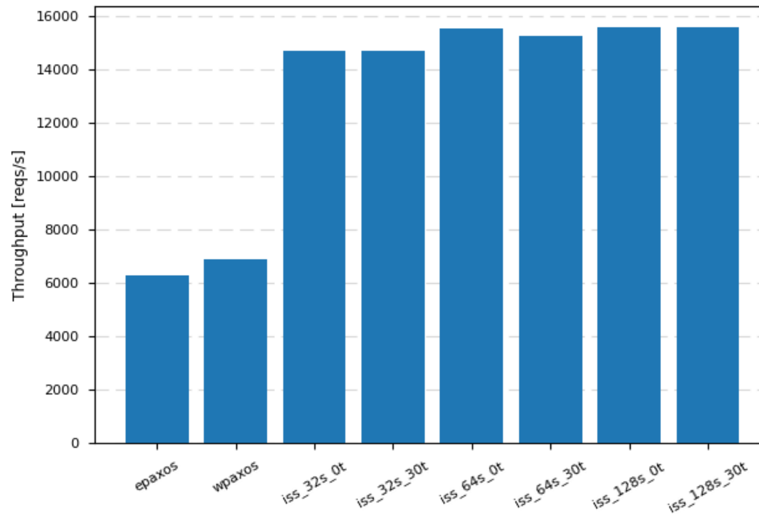


Figure 9. Maximum throughput for 3 nodes

As for the median latency, as can be seen in Figure 10, WPaxos and EPaxos obtained much better results, with values close to 1ms when processing 5400 requests per second. ISS, on the other hand, presented elevated latencies, from 14ms to 60ms, due to the fact that, besides the time necessary for a value to be decided, clients may also need to wait until the node to which their request was sent becomes the leader of the segment that has access to the bucket in which that request is stored. To corroborate this fact, it can also be seen that larger segments sizes, which take longer for rotating buckets, also present a

larger latency. As for the wait time, when not waiting for requests, the latency for requests is constant throughout the execution, due to the fact that proposing SKIP messages would have the same effect on latency as proposing an actual value. For nodes that waited 30ms, it can be seen that latency was increased due to longer epochs caused by nodes waiting for requests.

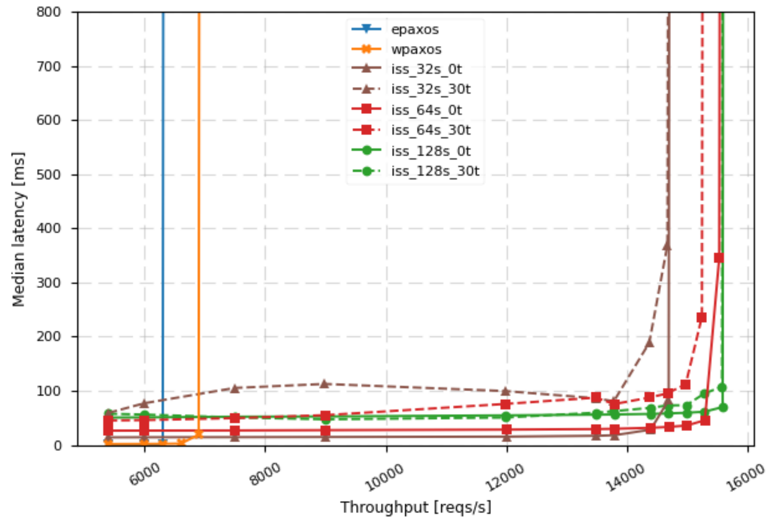


Figure 10. Median Latency for 3 nodes

Lastly, during experiments with three nodes, client behavior was slightly different for ISS nodes, with clients randomly selecting a server node for each request that had to be sent. This was done in order to accurately partition client nodes into buckets, but led to a random variance in the results for the 99 percentile latency, as can be seen in Figure 11. Due to this, in the other experiments, the behavior of ISS clients was changed to match the other algorithms, in which each client exclusively sends requests to a single server. For the buckets hashing algorithm, in order to simulate an even distribution of clients, requests were partitioned based on the number of the request, that is sequentially generated by clients.

4.4. 6 nodes

With the increase in number of nodes, there was an reduction in throughput on all algorithms, as can be seen in Figure 12. The greatest relative reduction can be seen in EPaxos, with a 14% reduction, followed by ISS, with a mean reduction close to 10%, and finally WPaxos, that only lost 4% of its maximum throughput. This could indicate that WPaxos has a better scalability than the version of ISS that was implemented, potentially presenting better throughput in situations with more participating nodes.

As for the median latency, as seen in Figure 13, EPaxos and WPaxos had a relatively small increase, not going past the 2ms while processing 5400 requests per second. ISS' latency, on the other hand, almost quadrupled due to, not only the increased number of messages needed for each decision, but also the fact that the increase in the number of nodes also makes it so epochs are bigger and clients need to wait more for bucket rotations.

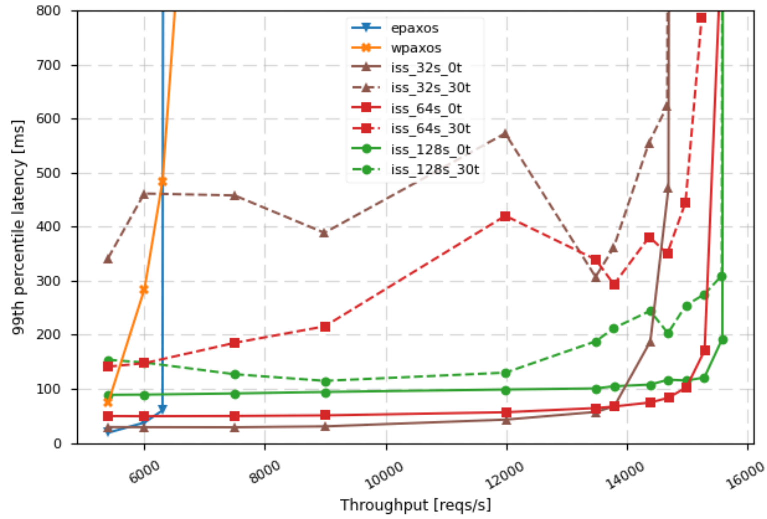


Figure 11. 99th Percentile Latency for 3 nodes

With more balanced client loads, it's also possible to properly see how nodes that wait for requests would behave. In both the median and 99 percentile cases, the latency in low throughput scenarios is higher than the non-waiting counterpart, due to taking a longer amount of time to rotate its buckets. As the throughput increases, the latencies of waiting variants begin to be similar or even better than non-waiting ones, due to the fact that waiting is done less often and, when done, is more likely to lead to an actual request being selected.

4.5. 6 nodes (SKIP Optimization)

Lastly, by optimizing the SKIP messages, such that those messages can be decided in a single P2A message, we can see in Figures 15 and 16 that although behavior of the ISS configurations remains relatively the same in high throughput situations, due to the fact that SKIP messages are rarely sent close to the saturation point of the protocol, the optimization can effectively reduce latency in lower thoughput scenarios. More precisely, latencies at 5400 requests per seconds, situation in which SKIP messages are more commonly sent, were reduced by approximately half. With the increase in throughput, latency then steadily increases as the number of SKIP messages being sent is reduced, until it reaches values close to that of the non-optimized version as practically no SKIP messages are sent at this stage..

5. Conclusion

In this work, we evaluated the throughput, scalability and latency of ISS, a framework capable of turning single-leader protocols into multi-leader ones, when applied to the Paxos protocols. In order to compare the results and approach of this framework when compared with other Paxos variants, as well as potentially find points of improvement, multiple Paxos variants were presented, of which WPaxos and EPaxos, both multi-leader variants, were selected for the comparison. During this evaluation, it was possible to determine that the implemented framework possess better throughput when replicating 3 and 6 nodes, but could potentially have a worst performance than the other variants in

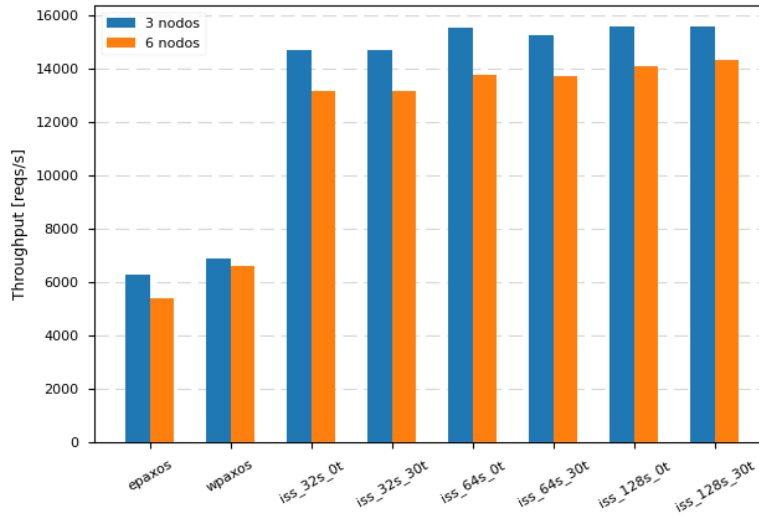


Figure 12. Maximum throughput comparison between 3 and 6 nodes

higher nodes configurations. The framework also presented considerably higher latency than other variants, due to its strategy of partitioning the client segment, making it so, if a request is sent to a single node, the client must wait a longer time until the epoch change for reconfiguration, making it so clients wait longer for an answer.

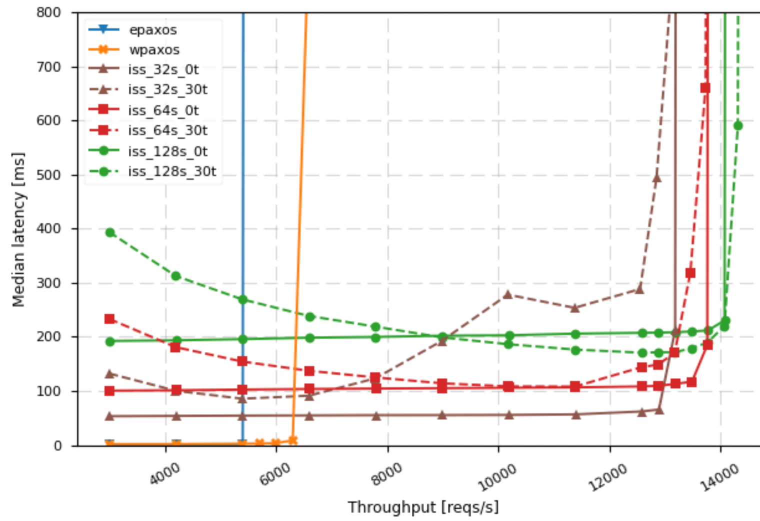


Figure 13. Median Latency for 6 nodes

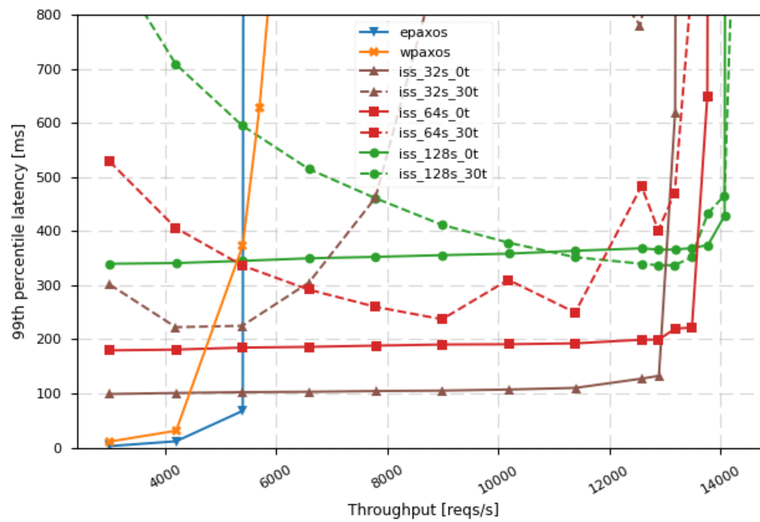


Figure 14. 99th Percentile Latency for 6 nodes

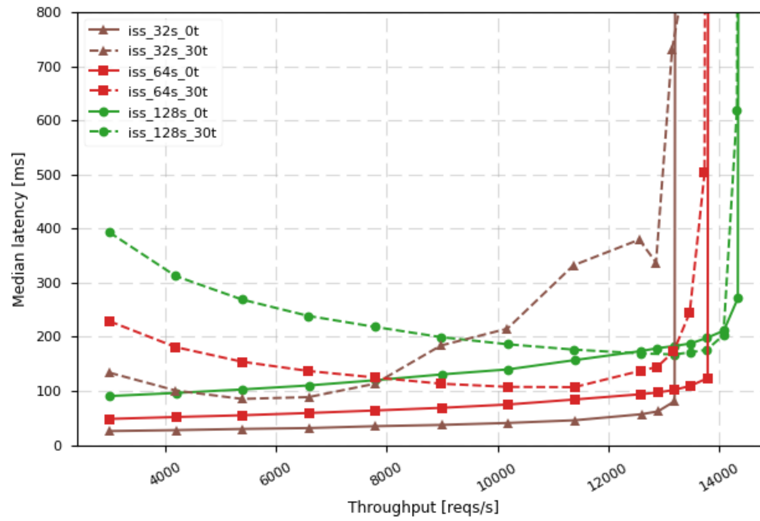


Figure 15. Median Latency for 6 nodes with SKIP optimization

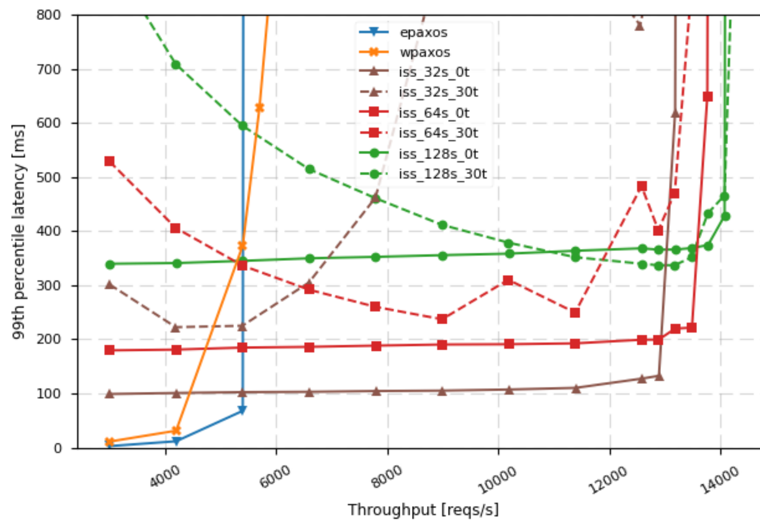


Figure 16. 99th Percentile Latency for 6 nodes with SKIP optimization

References

Ailijiang, A., Charapko, A., and Demirbas, M. (2019). Dissecting the performance of strongly-consistent replication protocols. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1696–1710.

Ailijiang, A., Charapko, A., Demirbas, M., and Kosar, T. (2020). Wpaxos: Wide area network flexible consensus. *IEEE Trans. Parallel Distrib. Syst.*, 31(1):211–223.

Cachin, C., Guerraoui, R., and Rodrigues, L. (2014). *Introduction to Reliable and Secure Distributed Programming*. Springer Publishing Company, Incorporated, 2nd edition.

Castro, M. and Liskov, B. (1999). Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, page 173–186, USA. USENIX Association.

Charapko, A., Ailijiang, A., and Demirbas, M. (2021). Pigpaxos: Devouring the communication bottlenecks in distributed consensus. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 235–247, New York, NY, USA. Association for Computing Machinery.

Howard, H., Malkhi, D., and Spiegelman, A. (2016). Flexible paxos: Quorum intersection revisited.

Jalili, P., Primi, M., Schiper, N., and Pedone, F. (2010). Ring paxos: A high-throughput atomic broadcast protocol. *Proceedings of the International Conference on Dependable Systems and Networks*, pages 527–536.

Lamport, L. (1998). The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169.

Lamport, L. (2001). Paxos made simple. *Sigact News - SIGACT*, 32.

Lamport, L. (2006). Fast paxos. *Distrib. Comput.*, 19(2):79–103.

Lamport, L., Malkhi, D., and Zhou, L. (2009). Vertical paxos and primary-backup replication. Technical Report MSR-TR-2009-63, Microsoft Research.

Lamport, L. and Massa, M. (2004). Cheap paxos. In *International Conference on Dependable Systems and Networks (DSN 2004)*.

Mao, Y., Junqueira, F. P., and Marzullo, K. (2008). Mencius: building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 369–384, USA. USENIX Association.

Moraru, I., Andersen, D. G., and Kaminsky, M. (2013). There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 358–372, New York, NY, USA. Association for Computing Machinery.

Ongaro, D. and Ousterhout, J. (2014). In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, page 305–320, USA. USENIX Association.

Stathakopoulou, C., Pavlovic, M., and Vukolić, M. (2022). State machine replication scalability made simple. In *Proceedings of the Seventeenth European Conference on*

Computer Systems, EuroSys '22, page 17–33, New York, NY, USA. Association for Computing Machinery.

White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A. (2002). An integrated experimental environment for distributed systems and networks. pages 255–270, Boston, MA.