

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Bruno Victor da Silva

**Otimizando a primeira fase do algoritmo Paxos na
biblioteca LibPaxos**

Uberlândia, Brasil

2020

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Bruno Victor da Silva

**Otimizando a primeira fase do algoritmo Paxos na
biblioteca LibPaxos**

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como requisito exigido parcial à obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Paulo Rodolfo da Silva Leite Coelho

Universidade Federal de Uberlândia – UFU

Faculdade de Computação

Bacharelado em Ciência da Computação

Uberlândia, Brasil

2020
Bruno Victor da Silva

Otimizando a primeira fase do algoritmo Paxos na biblioteca LibPaxos

Trabalho de conclusão de curso apresentado à
Faculdade de Computação da Universidade
Federal de Uberlândia, Minas Gerais, como
requisito exigido parcial à obtenção do grau de
Bacharel em Ciência da Computação.

Paulo Rodolfo da Silva Leite Coelho
Orientador

Lásaro Jonas Camargos

Rafael Pasquini

Uberlândia, Brasil

2020

Resumo

Chegar a um consenso é um problema fundamental para a consistência de sistemas distribuídos. Na replicação de máquina de estados, cada réplica precisa executar a mesma sequência de operações. Nesse contexto, consenso é utilizado para garantir que cada réplica decida a mesma sequência de operações. Além de um bom desempenho, um algoritmo de consenso deve ser capaz de proporcionar segurança e tolerar algumas falhas que podem acontecer durante sua execução para evitar a interrupção no funcionamento do sistema.

Este trabalho é desenvolvido sobre o algoritmo Paxos, um dos primeiros algoritmos criados para resolver este problema. Inicialmente é feito o estudo de uma biblioteca que implementa o algoritmo, denominada Libpaxos. A biblioteca é então otimizada de acordo com as melhorias apresentadas na literatura relacionada. Por fim, experimentos para análise de performance, confirmam os benefícios das melhorias introduzidas.

Palavras-chave: Sistemas distribuídos; protocolo de consenso; Paxos; otimização; replicação de máquina de estado.

Lista de ilustrações

Figura 1	Ũ <i>Multi-Paxos</i> com pré-execução já implementado pela LibPaxos	15
Figura 2	Ũ <i>Multi-Paxos</i> otimizado implementado neste trabalho	15
Figura 3	Ũ Exemplo de sistema bancário <i>sem</i> o uso de Multi-Paxos	17
Figura 4	Ũ Exemplo de sistema bancário <i>com</i> o uso de Multi-Paxos	17
Figura 5	Ũ Sequência de execução no cenário mais simples	23
Figura 6	Ũ Sequência de execução no cenário com interrupções	24
Figura 7	Ũ Comparação da Latência relacionado com o numero de clientes.	26
Figura 8	Ũ Comparação da vazão relacionado com o numero de clientes.	27
Figura 9	Ũ Comparação entre vazão e latência.	27

Sumário

1	INTRODUÇÃO	6
1.1	Objetivo	8
1.2	Metodologia	9
1.2.1	Análise	9
1.2.2	Compilação	9
1.2.3	Otimização	9
1.2.4	Avaliação	9
2	FUNDAMENTAÇÃO TEÓRICA	10
2.1	Paxos	10
2.1.1	Escolhendo o valor	11
2.1.2	Propagando o valor	13
2.2	Multi-Paxos e otimizações	14
2.2.1	Exemplo de aplicação	16
3	DESENVOLVIMENTO	18

3.1	Libpaxos	18	
3.1.1	Proposer	18	
3.1.2	Acceptor	19	
3.1.3	Learner	20	
3.1.4	Mensagens	20	
3.2	Alterações Propostas	21	
3.3	Cenários de Testes de Validação	22	
3.3.1	Mais simples	22	
3.3.2	Dois ou mais proposers	23	
4	EXPERIMENTOS	25	
4.1	Teste de Latência	26	
4.2	Teste de Vazão	26	
4.3	Comparação de latência e vazão	27	
5	CONCLUSÃO	29	
	REFERÊNCIAS		30

1 Introdução

A necessidade de sistemas que toleram falhas é muito comum em várias aplicações. Com isso surgem vários problemas que podem afetar o desempenho e o funcionamento de uma aplicação, como por exemplo, a falha de um dos servidores que executa alguma requisição feita por usuários e compõem parte de uma rede de outros processos sendo executados. Uma técnica usada para este tipo de caso é a replicação de máquina de estado (LAMPOR, 2005).

A replicação de máquina de estado tem um conceito bem simples que consiste em replicar as informações de um servidor para outro mantendo uma comunicação entre eles. Conforme as requisições vão sendo feitas, a sequência de comandos deve ser deterministicamente executada, ou seja, as réplicas fazem transições pela mesma sequência de estados (ESPOSITO; COELHO; PEDONE, 2018).

Essa técnica resolve o problema de falha, pois o usuário precisa receber apenas uma resposta de um dos servidores. Caso ocorra a falha de um servidor o sistema continuará operante pelo espelhamento feito na replicação até que o problema seja resolvido. A replicação de máquinas de estado envolve outro problema denominado consenso (LAMPOR, 1998).

Quando um sistema possui uma coleção de processos em execução, várias requisições podem ser aceitas simultaneamente por servidores diferentes, alterando a configuração da máquina de estado e deixando-os com informações diferentes. Um algoritmo de consenso deve garantir que um único valor proposto seja escolhido e, se nenhum valor é proposto, então nenhum valor é escolhido. Se um valor foi escolhido, os processos devem aprender o mesmo valor escolhido (PRISCO; LAMPSON; LYNCH, 1997).

Isso requer que algoritmos de consenso sejam criados para serem executados por todas as partes do sistema. Um dos mais importantes já criado foi o Paxos, publicado em 1998 (LAMPOR, 1998) por Leslie Lamport com várias melhorias e adaptações já publicadas como o *Fast Paxos* (LAMPOR, 2006) e o *Generalized Paxos* (LAMPOR, 2010). Esse protocolo separa os processos em três funções diferentes, os *proposers*, *acceptors* e os *learners*, garantindo segurança e eliminando a inconsistência entre eles (ESPOSITO; COELHO; PEDONE, 2018).

Os *proposers* são os usuários do sistema. Estes processos são responsáveis por enviar valores para serem decididos pelos *acceptors*, eles podem fazer várias requisições para que consigam ter um valor aceito. A execução do Paxos possibilita a escolha de apenas um valor em comum entre os diferentes servidores, esse acordo é chamado de instância, pois o algoritmo é executado várias vezes para que vários valores sejam decididos, e em cada

execução são feitas diferentes requisições, assim os *proposers* conseguem ganhar prioridade e ter sua proposta aceita para uma futura rodada.

Os *acceptors* formam grupos chamados de quóruns, toda requisição enviada para um *acceptor* deve ser enviada para um quórum de *acceptors*. A requisição só é aceita quando uma maioria de *acceptors* enviar uma resposta positiva para ela. Os *acceptors* garantem que se um valor já foi escolhido, nenhum outro valor poderá ser escolhido.

Os *learners* aparecem como fator de replicação do protocolo. Quando uma proposta for aceita, um *learner* deve descobrir que uma requisição foi aceita pela maioria dos *acceptors*. Para isso cada *acceptor*, sempre que aceitar uma requisição, deve responder a todos os *learners* enviando a requisição aceita para manter suas informações atualizadas (HOWARD; MALKHI; SPIEGELMAN, 2017).

O algoritmo funciona em duas fases. Na primeira fase o *proposer* tenta obter a promessa de uma maioria dos *acceptors*. Na fase seguinte o *proposer*, caso tenha obtido a promessa da maioria dos *acceptors* na primeira fase, envia o valor que ele deseja propor. Com a resposta positiva da maioria dos *acceptors* na segunda fase, o valor é enviado aos *learners*. Um mesmo processo pode assumir múltiplos papéis no Paxos (BOICHAT et al., 2003).

Desse modo, a replicação de máquina de estado pode ser implementada como uma sequência de instâncias independentes de Paxos. Cada instância decide um comando a ser executado. Os *learners* executam os comandos de cada instância em uma ordem determinística para garantir que passem pelas mesmas transições de estado. Essa abordagem é denominada como Multi-Paxos.

Em um cenário típico, as réplicas do sistema incorporam os três papéis do Paxos. Deste modo, o sistema continua operante enquanto uma maioria simples das réplicas não falhar. Isto significa que, para tolerar a falha de uma réplica, o sistema precisa de ao menos três réplicas (ESPOSITO; COELHO; PEDONE, 2018).

LibPaxos é uma implementação de Multi-Paxos feita na linguagem C. Seu código está disponível publicamente no bitbucket.org. A biblioteca separa a lógica do algoritmo da infraestrutura de rede, facilitando a extração e inclusão de novos recursos.

O presente trabalho visa estender a biblioteca para incluir as otimizações propostas em *Stoppable Paxos*, executando simultaneamente a fase de obtenção da promessa para todas as instâncias, enviando uma única mensagem para cada *acceptor*. Quando a mensagem chega, o *acceptor* executa simultaneamente as respostas para todas as instâncias, agrupando o conjunto de mensagens em uma única mensagem física (LAMPORT, 1998).

1.1 Objetivo

Este trabalho visa fazer alterações na biblioteca LibPaxos, a qual implementa a execução do algoritmo Paxos dentro de um sistema distribuído, seguindo as propostas apresentadas em *Stoppable Paxos* (LAMPORT; MALKHI; ZHOU, 2008), a fim de otimizar o funcionamento do algoritmo.

Como os *proposers* podem fazer várias propostas a qualquer momento executando as duas fases do algoritmo, sempre que uma proposta for rejeitada na primeira fase a segunda fase não será executada, evitando a troca de mensagens desnecessárias. Geralmente, a menos que haja falha, o mesmo *proposer* executa instâncias sucessivas do Paxos. Desse modo, ele pode executar a primeira fase apenas na primeira instância e os *acceptors* consideram o *ballot* da primeira instância para as seguintes. Com isso, as requisições são decididas em um *round-trip*, aumentando o *throughput* no sistema.

Após a implementação e execução das mudanças propostas no *Stoppable Paxos*, serão realizados testes no algoritmo para comparar os resultados alcançados e analisar a diferença obtida entre a biblioteca original e a otimizada.

1.2 Metodologia

Para o desenvolvimento do trabalho será utilizada a versão mais recente da biblioteca LibPaxos juntamente com a revisão bibliográfica do artigo *Stoppable Paxos*. Esse artigo propõe otimizações no algoritmo conforme descrito na Seção 1.1. Para chegar ao objetivo do trabalho a metodologia foi dividida em quatro partes para que análises pudessem ser feitas separadamente.

1.2.1 Análise

Entender o mecanismo descrito em *Stoppable Paxos* e estudar o código e o funcionamento da biblioteca LibPaxos.

1.2.2 Compilação

Reproduzir a execução da biblioteca LibPaxos3, e entender como funciona a estrutura interna do código, com a simulação verificar a interação das mensagens que aparecem no terminal.

1.2.3 Otimização

Inicialmente será feita a alteração do código original desta biblioteca implementando as alterações propostas.

1.2.4 Avaliação

Simular o projeto em uma rede com *proposers*, *acceptors* e *learners* em diferentes máquinas. Medir a latência para um cliente e a quantidade de valores enviados máxima aumentando o número de clientes até a saturação do sistema. Fazer testes com simulação de falha de *proposers* e uma avaliação de desempenho comparando a LibPaxos original com a LibPaxos otimizada, para analisar os resultados obtidos no algoritmo original e no otimizado.

2 Fundamentação Teórica

Neste capítulo será apresentado detalhadamente o funcionamento do algoritmo Paxos proposto por Lamport, bem como algumas otimizações propostas para o mesmo.

2.1 Paxos

O objetivo do algoritmo é solucionar o problema de consenso existente entre aplicações distribuídas. Basicamente esse problema se resume em chegar a uma decisão comum a todos considerando algumas propriedades, independente das propostas de cada um. Ao final da execução, uma ação será única e replicada em todos os dispositivos, tolerando possíveis falhas de funcionamento nos agentes (LAMPSON, 2001).

O sistema funciona inicialmente através da solicitação de alguma proposta feita indiretamente por um usuário ou cliente da aplicação, mais detalhadamente essa proposta deve ser encaminhada a um *proposer* pois somente eles são responsáveis por preparar essa requisição ao quórum de *acceptors*. Para chegar ao consenso é necessário que:

- Um valor só possa ser escolhido caso ele tenha sido proposto;
- A proposta aceita seja única;
- A proposta seja replicada somente após ser aceita;

O *proposer* tem o papel de preparar novas propostas e enviar para os *acceptors*, solicitando que algum valor seja aceito, ou para que ele tenha permissão de enviar algum valor para ser aceito. Já o *acceptor* é responsável por receber propostas e decidir se elas serão aceitas ou rejeitadas de acordo com as propriedades do algoritmo. Por fim, o *learner* representa a etapa de replicação do protocolo; após um valor ser acordado entre os *acceptors*, ele atualiza as informações para manter a consistência do sistema.

Um nó pode ter várias funções dependendo da sua ação no sistema. Para um valor ser aceito a maioria dos *acceptors*, deve dar seu reconhecimento. Paxos funciona apenas se a maioria dos *acceptors* não falharem simultaneamente. Caso essa condição não seja satisfeita ocorre um travamento, pois nenhum *proposer* poderá obter a aprovação para que seu valor seja aceito (BOICHAT et al., 2003).

Para a proposta ser aceita a maioria dos *acceptors* deve dar uma resposta de aceitação, pois se cada *acceptor* aceitar um valor diferente ele não será único. Para o caso em que é necessário a maioria, para a proposta ser aceita ela terá pelo menos um *acceptor* em comum entre rodadas consecutivas.

Para escolher o valor, alguns casos foram analisados, a maneira mais simples é tendo somente um *acceptor*, o *proposer* envia uma proposta e cabe a ele a decisão de escolher o valor proposto. Caso ele falhe o sistema ficaria inoperante, então seria melhor considerar o caso de um conjunto de *acceptors*, onde os *proposers* enviam valores para uma combinação desse grupo, e esse valor seria aceito caso a maioria dos *acceptors* o aceitassem (LAMPORT, 1998).

A troca de mensagens entre os componentes segue o modelo assíncrono e não bizantino, ou seja, pode haver atrasos de respostas, mensagens perdidas ou duplicadas e ainda os próprios agentes podem falhar. A comunicação é feita por meio de *buffers* responsáveis por armazenar mensagens recebidas e a serem enviadas que não podem ser corrompidos (CASON et al., 2017).

2.1.1 Escolhendo o valor

A execução do algoritmo para decidir um valor pode ser dividida em duas fases. Na primeira fase o *proposer* envia um valor com o *ballot* desejado, solicitando a permissão para um grupo de *acceptors*, e os *acceptors* garantem rejeitar outras solicitações caso um *ballot* maior já tenha sido aceito em outras propostas anteriores retornando ao *proposer* o valor aceito e o valor do *ballot* solicitado em outra mensagem (CHANDRA; GRIESEMER; REDSTONE, 2007).

O valor é escolhido caso a maioria dos *acceptors* retornem uma resposta de aceitação para a proposta, para apenas um valor ser escolhido esse conjunto deve ser qualquer maioria entre os agentes. Entretanto queremos que um valor seja escolhido mesmo que apenas um valor seja proposto (LAMPORT et al., 2001), isso requer a seguinte definição:

Def. P1: Um *acceptor* deve aceitar a primeira proposta que receber.

Mas se cada *acceptor* aceitar a primeira proposta que chega até ele vindo de diferentes proponentes, isso pode ocasionar uma discordância entre eles. Para se chegar ao consenso um único valor deve ser decidido, isso implica que um *acceptor* pode aceitar mais de uma proposta em uma determinada rodada de execução.

Os *proposers* enviam propostas contendo um número de identificação chamado de *ballot*, esse valor é único em cada proposta criada e não pode ser repetido na mesma rodada de execução, esse valor vai crescendo monotonicamente em cada *proposer* caso haja necessidade. Podemos permitir a escolha de várias propostas desde que garantimos que elas possuem o mesmo valor, então garantimos que:

Def. P2: Se uma proposta for escolhida, toda proposta de *ballot* mais alto escolhida terá o mesmo valor.

Como os *ballots* podem ser ordenados, a propriedade de se escolher apenas um valor é garantida. Para uma proposta ser escolhida pelo menos um *acceptor* deve retornar uma mensagem de aceitação, assim:

Def. P2a: Se uma proposta for escolhida, toda proposta de *ballot* mais alto escolhido por qualquer *acceptor* terá o mesmo valor.

Podemos encontrar casos em que a definição P2a pode não ser cumprida mantendo a definição P1, por exemplo: como a comunicação é assíncrona, uma proposta pode ser aceita por um *acceptor* *c* que nunca tenha recebido uma proposta, suponha que um novo *proposer* seja criado e envia uma proposta de *ballot* mais alto com um valor diferente para este *acceptor* *c*. Isso exige que *c* aceite essa proposta violando P2a. Isso requer o fortalecimento de P2a, com a seguinte afirmação:

Def. P2b: Se uma proposta for escolhida, toda proposta de *ballot* mais alto enviada por um *proposer*, terá o mesmo valor.

Como as propostas precisam ser feitas pelos *proposer* para que algum *acceptor* possa aceitá-la, a definição P2b implica em P2a, que por Δm implica em P2, para garantir ainda mais P2 temos que:

Def. P2c: Para qualquer proposta (n, v) enviada, então há um quórum *S* de *acceptors* em que:

1. Nenhum *acceptor* pertencente a *S* aceitou qualquer proposta com *ballot* menor.
2. v é o valor da proposta de *ballot* mais alto entre todas as propostas numeradas menor que *n* aceitas por *S*.

Qualquer um dos casos de P2c satisfaz P2b. Para isso sempre que *proposer* for emitir uma nova proposta ele deve ser capaz de aprender a proposta de maior *ballot* se o seu for menor, caso já exista alguma aceita por algum quórum. Para isso, o *proposer* envia uma solicitação aos *acceptors* para que eles não aceitem uma nova proposta de *ballot* menor, seguindo o seguinte algoritmo:

1. Um *proposer* escolhe um *ballot* e envia a cada membro do quórum solicitando que ele não aceite uma promessa de *ballot* menor que o seu, e envie a proposta com maior *ballot* menor que o seu próprio se houver.
2. Se os *acceptors* responderem às solicitações, o *proposer* emite sua proposta (n, v) sendo v o valor da proposta e n o número mais alto entre as respostas. Caso os *acceptors* não tenha relatado propostas, o *proposer* escolhe qualquer valor.

Com esse algoritmo percebemos que um *proposer* envia uma proposta do tipo *prepare* ao quórum de *acceptors* esperando uma resposta caso já não tenha sido aceito nenhuma

outra. Em seguida ele envia outra proposta do tipo *accept*, não necessariamente para o mesmo quórum, solicitando que a proposta seja aceita.

Já os *acceptors* podem receber solicitações de *prepare* e solicitações de *accept*. Um *acceptor* pode ignorar qualquer solicitação sem comprometer a segurança e o funcionamento do algoritmo. Portanto, precisamos dizer apenas quando é permitido responder a uma solicitação.

Com essa proposta, os *acceptors* devem enviar uma mensagem do tipo *accepted* para o *proposer*. Um *proposer* pode fazer e abandonar novas propostas a qualquer momento, isso quer dizer que:

Def. P1a: Um *acceptor* só pode aceitar uma proposta (n, v) , somente se, ainda não tiver respondido uma solicitação *prepare* com um identificador maior que n .

Seguindo essas regras garantimos a propriedade de ter números únicos para as propostas. Ainda existe uma maneira de otimizar este mecanismo, o *acceptor* pode ignorar novas propostas quando receber solicitações de *prepare* com *ballot* menor do que já foi decidido, também pode ignorar solicitações de *prepare* que já tenha aceitado.

O *acceptor* precisa guardar apenas a proposta com o maior *ballot* aceito e a proposta com *ballot* mais alto a qual já respondeu, essas informações não podem ser perdidas e devem ser possíveis de serem recuperadas. Com essas informações podemos melhor definir as duas fases citadas anteriormente da seguinte maneira:

Fase 1

a. O *proposer* escolhe um *ballot* e envia uma solicitação de *prepare* para um quórum de *acceptors*.

b. O *acceptor* recebe a solicitação de *prepare* e verifica o *ballot* recebido, se ele é maior do que qualquer outro já respondido, ele responde à solicitação com a promessa de não aceitar outras propostas com *ballot* menor, junto com a proposta de *ballot* mais alto, caso já tenha aceito outra.

Fase 2

a. Se o *proposer* receber uma promessa da maioria do quórum para um determinado *ballot*, ele envia uma solicitação de *accept* a cada um desses *acceptors* com uma proposta (n, v) , sendo n o *ballot* recebido e v o valor da proposta

b. O *acceptor* recebe uma solicitação de *accept* e aceita caso não tenha recebido outra solicitação *prepare* com *ballot* maior, do contrário a solicitação é negada para dar preferência ao *proposer* que realizou a nova solicitação de *prepare*.

2.1.2 Propagando o valor

Para saber se uma proposta foi aceita, os *learners* devem descobrir se ela recebeu uma resposta da maioria dos *acceptors*. Existem diversas maneiras de se implementar essa etapa, e fica a critério do programador (LAMPORT et al., 2001).

Um modo mais fácil é fazer com que cada *acceptor*, sempre que aceitar uma proposta, envie uma mensagem aos *learners* informando a proposta, assim cada *acceptor* deve responder a cada *learner*.

O *learner* pode descobrir de outros *learners* a proposta aceita. Uma alternativa é fazer com que os *acceptors* respondam somente a um *learner* específico, chamado de *distinguished learner*, e a tarefa de replicar a proposta entre os demais seria somente dele, isso requer uma nova etapa de troca de mensagens após a fase 2. Também há um problema de confiabilidade, pois se o *distinguished learner* (LAMPORT; MASSA, 2004) falhar o sistema ficará inconsistente, porém requer uma troca de mensagens igual a soma do número de *acceptors* e o número de *learners*.

2.2 Multi-Paxos e otimizações

Na prática as aplicações que precisam resolver o problema de consenso executam o protocolo Paxos várias vezes, pois a cada execução só é possível decidir um valor proposto e normalmente não é esse o objetivo a ser alcançado. Esse tipo de estratégia é chamado de *Multi-Paxos* (ESPOSITO; COELHO; PEDONE, 2018). Com isso podemos discutir várias ideias para otimizar seu funcionamento que antes não eram previstas no algoritmo básico.

Uma destas otimizações já utilizada pela biblioteca é a pré-execução da fase 1. Isso significa que, durante a inicialização do *proposer*, ele realiza fase 1 para uma certa quantidade de instâncias, e assim que essas instâncias forem sendo utilizadas um novo grupo de instâncias é pré executada. Nesta biblioteca esse procedimento é feito com uma variável que verifica a quantidade de instâncias pré executadas no armazenamento formando um tipo de janela de atualização, criando um esquema como na Figura 1.

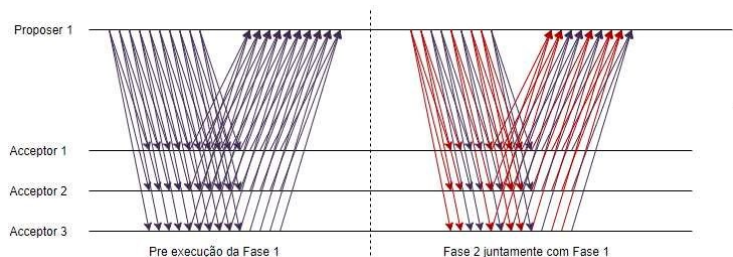


Figura 1 *Multi-Paxos* com pré-execução já implementado pela LibPaxos.

A ordem de solicitações e respostas não necessitam estar em sequência e ordenadas como mostrado na Figura 1, haja visto que as mensagens são assíncronas, estão assim apenas para facilitar o entendimento.

Outra otimização visa reduzir o número de mensagens entre os agentes, realizando a fase 1 apenas uma única vez, sendo a otimização implementada neste trabalho. Caso o proposer líder, o qual é responsável por fazer as propostas aos *acceptors*, não seja interrompido, é feita apenas a fase 2 sucessivamente pelo mesmo líder enquanto um outro não enviar uma proposta com *ballot* maior, conforme Figura 2.

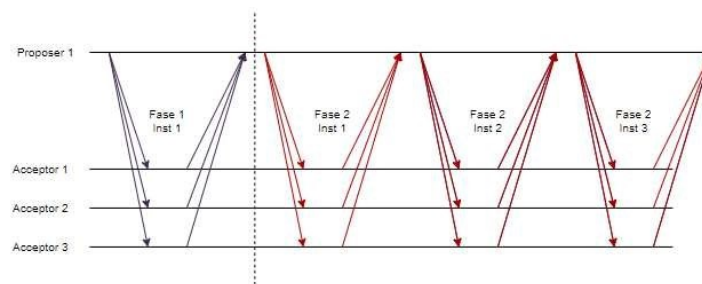


Figura 2 *Multi-Paxos* otimizado implementado neste trabalho.

A linha tracejada representa o ponto de criação do *client* e início das requisições. Para fazer isso, o *acceptor* precisa informar ao *proposer* qual a maior instância e *ballot* aceitos por ele durante a fase 1, para que o *proposer* faça fase 1 até a instância informada com um *ballot* mais alto que o último aceito. Para que, caso aconteça alguma interrupção, outro *proposer* assuma esse papel e continue seu funcionamento normalmente.

Uma outra otimização proposta por Lamport é o *Fast Paxos* (LAMPORT, 2006), ele reduz o atraso entre a criação de um valor pelo *client* até a sua aceitação. No algoritmo original para que o valor seja aceito é preciso realizar as duas fases e após isso o *proposer* precisa enviar o valor aceito para os *learners* totalizando 3 atrasos.

Com o algoritmo otimizado, os *acceptors* responderiam diretamente para os *learners* totalizando 2 atrasos na mensagem, e também se o *proposer* líder não tiver valor para propor, o *client* pode enviar uma solicitação de aceitação diretamente para os *acceptors*. Caso a proposta seja aceita, os *acceptors* enviariam uma mensagem ao líder e aos *learners*.

Caso ocorra colisão, o líder deve enviar uma solicitação de *accept* para a próxima rodada de execução como acontecia no algoritmo original. Essa recuperação coordenada de propostas requer 4 atrasos nas mensagens do *client* para os *learners* diminuindo o desempenho com o aumento de concorrência entre propostas.

Com isso é preciso utilizar uma técnica de recuperação antecipada, fazendo com que os *acceptors* realizem a recuperação de colisão. Nessa recuperação descoordenada ocorrem também 3 atrasos, ou 2, caso não aconteçam colisões. O *Fast Paxos* exige que o sistema possua $3f+1$ *acceptors* para tolerar f falhas como uma desvantagem, contra $2f+1$ no algoritmo original.

2.2.1 Exemplo de aplicação

Um bom exemplo para se compreender o algoritmo é imaginar o funcionamento de um sistema bancário, por exemplo, os clientes do sistema serão os caixas eletrônicos por onde as pessoas fazem consulta de saldo, pagamentos, saques entre outras operações. Cada operação que alterar o saldo de uma conta poderia ser uma conAguuração da máquina de estado e que seria replicada para todo o sistema.

Neste cenário, imaginemos um sistema com 3 servidores replicados, ou réplicas, cada um desempenhando os papéis de *proposer*, *acceptor* e *learner*. Esta é uma conAguuração típica, pois agrupa os papéis e tolera a falha de até uma réplica. Isto signiAca que o sistema bancário consegue operar normalmente com duas réplicas ativas em um dado momento. A réplica que falhou pode recuperar o estado a partir do disco ou de outra réplica antes de reiniciar sua operação. Cada réplica parte do mesmo estado inicial e uma delas é escolhido como *proposer* líder.

No momento da operação, o caixa eletrônico utilizando estaria realizando o papel do *cliente*, supondo que um saque será feito, este terminal estaria fazendo a solicitação para a réplica deAñida como líder descontar um valor na conta somente se o saldo for maior que a quantia sacada. Após obter a resposta positiva de uma maioria de réplicas (que são também *acceptors*), incluindo o próprio líder, é possível retornar e retornar uma resposta para o usuário.

Neste caso, solicitações de saque concorrentes serão propostas pelo líder em instâncias sucessivas e executadas pelo *learner* de cada réplica na mesma ordem, mantendo o estado consistente.

Considere um exemplo em que três réplicas possuem apenas uma conta de saldo inicial igual a 100 e que esta conta não pode haver saldo negativo. Caso as réplicas não utilizem nenhum protocolo de consenso, a situação apresenta na Figura 3 pode ocorrer: clientes solicitam operações de saque (S) a réplicas diversas deixando o estado Añal inconsistente pois uma das réplicas possui saldo divergente das demais.

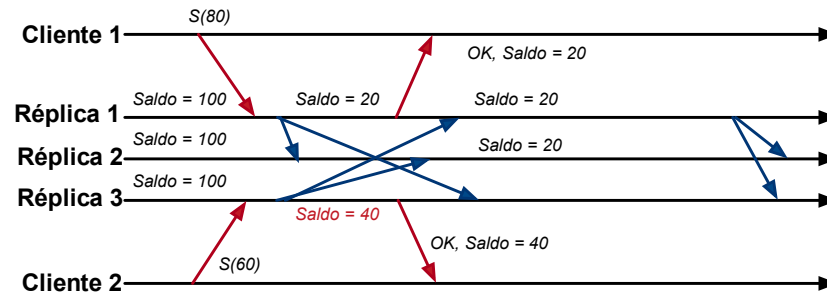


Figura 3 $\tilde{U}$ Exemplo de sistema bancário *sem* o uso de Multi-Paxos.

Na Figura 4 vemos o que acontece quando utilizamos Multi-Paxos. Mesmo que os clientes acessem réplicas diferentes, as requisições são encaminhadas ao líder, o qual propõe cada solicitação em uma instância sucessiva (10 e 11, neste caso), fazendo com que o estado das réplicas evolua da mesma maneira.

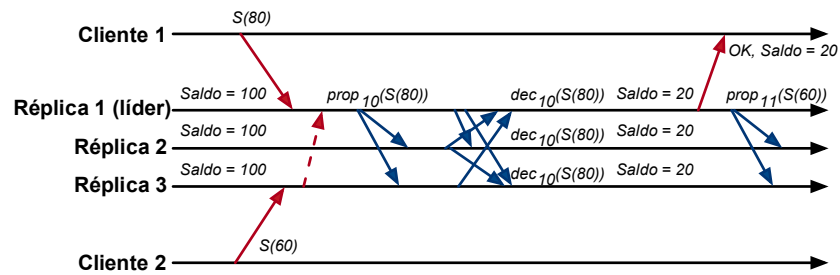


Figura 4 $\tilde{U}$ Exemplo de sistema bancário *com* o uso de Multi-Paxos.

3 Desenvolvimento

Nesta seção será apresentada a biblioteca LibPaxos3 e quais foram as alterações feitas na biblioteca original para se atingir o objetivo proposto na Seção 1.1. Para cada alteração realizada, será explicado o que isso altera no funcionamento do *Multi-Paxos*. A implementação do código otimizado está disponível de forma aberta no Bitbucket ¹.

3.1 Libpaxos

Neste trabalho foi utilizado a biblioteca *Libpaxos3* ² para simular a execução do Paxos. Ela é uma reescrita da biblioteca *LibPaxos2*, e se divide em duas bibliotecas: a *libpaxos/paxos* que implementa as propriedades do Paxos e a *libpaxos/evpaxos* que implementa parte de rede entre os agentes, construída sobre a *libpaxos* utilizando a biblioteca *libevent* ³.

Esta biblioteca é implementada em linguagem C, sendo executável em sistemas operacionais *Linux* e *OS X*. Para executá-la, é preciso instalar pacotes auxiliares como o *build-essentials* e *cmake* para compilar o projeto, *libevent-dev* para notificação de eventos assíncronos provendo um mecanismo de resposta, *msgpack-dev* para serializar as mensagens na rede e opcionalmente o *lmdb* para auxiliar na persistência em disco de estado dos *acceptors*.

Na execução, os agentes utilizam um *ID* para se diferenciar uns dos outros, um endereço *IP* e portas para se conectarem. Elas podem ser escolhidas alterando o arquivo *paxos.conf*. Para facilitar o entendimento da biblioteca ela será dividida entre seus agentes.

3.1.1 Proposer

O funcionamento de rede do *proposer* está implementado no arquivo *evproposer.c* na biblioteca *evpaxos*. Inicialmente ele é criado na função *evproposer_init* e após estabelecer conexão com os *acceptors* ele pré executa a fase 1A 128 vezes. Isso quer dizer que ele envia uma requisição do tipo *prepare* para os *acceptors* pedindo para não aceitarem propostas com *ballots* menores durante as 128 próximas instâncias. Essas solicitações são controladas pela variável *count* dentro da função *proposer_preexecute*, a qual é decrementada sempre que é enviada uma solicitação de *accept*, fazendo com que o *proposer* sempre tenha uma janela de solicitações do tipo *prepare* igual a 128, de modo semelhante do apresentado na Figura 1.

¹ <https://bitbucket.org/brunoufu/brunoufu2020/src/master>

² <https://bitbucket.org/sciascid/libpaxos/src/master/>

³ <https://libevent.org/>

Em seguida, aguarda uma resposta da fase 1 de cada *acceptor*. Quando essa resposta chega, ele executa a função *evproposer_handle_promise* para verificar o tipo de mensagem, que pode ser um *preempted* significando que a proposta não foi aceita, ou pode ser um *paxos prepare* significando que ele pode enviar uma solicitação de *accept* para aquela instância, assim que o quórum de configuração for atingido.

Quando um *client* é iniciado, o *proposer* começa a receber valores. Nessa etapa estamos apenas discutindo a camada de rede então o *proposer* sempre verifica se existe um valor para ser enviado através da função *evproposer_handle_acceptor_state* que é executada sequencialmente. Para realizar a fase 2, sempre que um determinado *proposer* envia ou recebe mensagens a função *try_accept* é executada, é ela que verifica se existe a possibilidade de se enviar uma mensagem 2A.

Quando uma solicitação de *accept* é enviada, o *acceptor* pode retornar uma resposta do tipo *preempted* significando que ela não foi aceita, retornando o processo novamente para a fase 1, ou ela pode ser do tipo *paxos_accepted* significando que a proposta foi aceita. Essas respostas são verificadas na função *evproposer_handle_accepted*, atendendo as propriedades do paxos explicadas anteriormente.

Todas essas funções citadas se comunicam diretamente com a biblioteca que implementa o protocolo em si. No caso do *proposer*, está implementado no arquivo *proposer.c* na biblioteca *paxos*. Lá se encontram as implementações de importantes tarefas utilizadas, como, por exemplo, a criação de instâncias. Sempre que uma fase 1 é realizada, uma nova instância é criada pela função *proposer_prepare*.

3.1.2 Acceptor

A organização do código do *acceptor* é semelhante ao do *proposer*, sua implementação também está separada nas duas bibliotecas, a parte de rede do *acceptor* está implementada no arquivo *evpacceptor.c* na biblioteca *evpaxos*. Inicialmente, ele é criado na função *evacceptor_init* e fica aguardando algum evento acontecer.

Como explicado anteriormente, o *acceptor* deve receber uma solicitação de *prepare*, para determinada instância, essa solicitação é tratada na função *evacceptor_handle_prepare* responsável por verificar se a proposta é válida e armazenar ou não aquela informação. Dependendo do que for decidido, uma mensagem é preparada e encaminhada na fase 1, e as informações são alteradas.

Caso o *acceptor* receba uma proposta de *accept*, ela será tratada na função *evacceptor_handle_accept*, responsável por verificar no armazenamento se deve enviar uma mensagem de aceitação ou não. As informações desse armazenamento estão no arquivo *storage.c* na biblioteca *paxos*. Por ele são acessadas informações necessárias para atender as

propriedades do paxos armazenadas no *backend* pelos arquivos *storage_lmdb.c* e *storage_mem.c*, onde são mantido as propostas aceitas.

As funções subordinadas do *acceptor* pela camada de rede estão implementadas no arquivo *acceptor.c*, na biblioteca *paxos*. Lá estão as funções responsáveis por consultar a validade das propostas no armazenamento como já citado, além de construir as mensagens que ele deve enviar nas funções *paxos_accepted_to_promise* durante a fase 1, *paxos_accept_to_accepted* durante a fase 2 e *paxos accepted to preempted* em caso de não aceitação.

3.1.3 Learner

O funcionamento de rede do *learner* está implementado no arquivo *evlearner.c* na biblioteca *evpaxos*. Inicialmente, ele é criado na função *evlearner_init*. O *learner* serve para replicar as informações na fase 2B, sendo assim, ele executa a função *evlearner_handle_accepted*. Quando um *accept ack* é recebido, o *learner* atualiza seu *status* para essa instância e depois verifica se a instância está fechada, ou seja, se recebeu *accept_ack* de uma maioria de *acceptors*.

As funções usadas no *learner* pela camada de rede estão implementadas no arquivo *learner.c* na biblioteca *paxos*, na qual estão algumas funções para manipular instâncias, como criar e deletar, verificar se foram aprendidas, guardar as instâncias aceitas no armazenamento, incrementar o identificador das instâncias e verificar se o quórum foi atingido.

3.1.4 Mensagens

A troca de mensagens entre os agentes são feitas utilizando *buffer* capaz de armazenar diferentes estruturas, que representam diferentes mensagens. Esses *buffers* estão presentes na biblioteca de rede *evpaxos* representados no arquivo *peers.c*. Além disso, toda mensagem ao ser recebida ou enviada é colocada lá para ser processada. Os processos conectam-se uns aos outros e trocam mensagens utilizando a biblioteca *libevent*.

Antes de uma mensagem ser enviada, ela precisa ser preparada utilizando a biblioteca *msgpack*. A implementação desta etapa está no arquivo *message.c* na pasta *evpaxos*. Dependendo do tipo de mensagem a ser enviada, uma determinada função é executada para encapsular os dados com seu respectivo rótulo. O envio da mensagem é feito na função *send_paxos_message* que utiliza tarefas implementadas no arquivo *paxos_types_pack.c*, no qual verifica-se o rótulo das mensagens para empacotamentos e desempacotamentos e serializa-os na rede. As informações presentes em cada tipo de mensagem estão definidas no arquivo *paxos_types.h* no diretório *include*.

3.2 Alterações Propostas

As alterações principais foram feitas na biblioteca *evpaxos*, responsável pela parte de rede, por onde o *proposer* recebe e envia mensagens para o *acceptor*. Como dito na Seção 3.1.1, o *proposer* pré executa 128 instâncias, o que não será mais necessário, pois a ideia é realizar a fase 1 apenas uma única vez. Isso é feito na função *proposer_preexecute*, que será utilizada apenas para realizar a fase 1, ou seja, ao ser inicializada ou caso outro *proposer* envie uma proposta com *ballot* maior.

Com isso a variável *count*, que representava inicialmente a quantidade de préexecuções da fase 1 pendentes, agora representa a quantidade de instâncias para as quais a fase 1 deve ser executada. Isso quer dizer que, se outro *proposer* enviou uma proposta com *ballot* mais alto e já teve valores aceitos para instâncias futuras, o *proposer* atual saberá que está atrasado através da variável *atual*, criada para guardar o *ID* da instância onde foi realizada a fase 1. Caso o *proposer* verifique que está atrasado, ele utiliza a variável *atraso* para realizar a função *pre execute*.

Neste caso, ao realizar a função *proposer_preexecute*, o *proposer* realiza a fase 1 até a maior instância aceita, já calculada pelo *count*, e atualiza as variáveis para que o *proposer* saiba que ele não está mais atrasado com relação aos demais *proposers*.

E preciso fazer o mesmo para o caso em que o *proposer* recebe uma mensagem de *preempted*. A função *evproposer_handle_preempted* é responsável por tratar este caso, nela são atualizados as variáveis assim como na função *proposer_preexecute*, mas não é preciso criar uma nova instância nesse caso basta enviar uma mensagem para a mesma instância com o próximo *ballot*.

Esse método de atualização das variáveis evita o problema da criação de mais de uma instância, porque assim que a primeira resposta é tratada pelo *proposer*, as variáveis são atualizadas e as demais respostas vão ser executadas com os novos valores. Esse problema acontece porque quando uma proposta é criada e enviada, o *proposer* irá receber um certo número *n* de respostas, sendo *n* a quantidade de *acceptors* criadas, isso quer dizer que ele irá executar a função *evproposer_handle_accepted* *n* vezes.

Com isso, o *proposer* entenderá que a instância foi aceita caso a maioria (quórum) de *acceptors* ($\frac{n}{2} + 1$), envie uma resposta do tipo *accepted*. Isso é verificado pela função *proposer_receive_accepted*, logo tudo que estiver nesta função será executado *n* vezes. No código original essa função indica o fim de uma instância do paxos, e marcava o início do próxima instância.

Ao ser executada, agora ela verifica a variável *atraso* e com isso indica o curso de execução do *proposer*, caso a variável *atraso* não seja alterada a execução fica em um *loop* entre criar nova instância e enviar uma solicitação de aceitação, feitos nas funções *try_accept*

e *evproposer_handle_accepted*, evitando a execução de outras partes do código. Caso algum *proposer* tenha feito uma proposta e consiga a prioridade, seu fluxo de execução é direcionado para realizar a fase 1 conforme dito anteriormente, ou seja, ela indica o que deve ser feito a cada resposta recebida para iniciar a próxima instância.

Ainda no *proposer*, dentro da biblioteca *paxos*, foi alterada a forma como o *ballot* era escolhido para que, quando um *ballot* dentro do mesmo *proposer* fosse utilizado, esse valor fosse comparado com o anterior e fosse escolhido sempre o maior, e esse valor se mantivesse para as instâncias futuras. O maior *ballot* aqui guardado na variável *ballot* juntamente com as informações do *proposer*.

Foi preciso incluir algumas informações no funcionamento do *acceptor*. Uma delas é informar a maior instância aceita aos *proposers*, para isso, foi incluído a variável *next* durante a fase 1, na mensagem *paxos promise*, e com isso foi necessário também serializar e desserializar as mensagens corretamente.

Nas informações do *acceptor* também foram adicionadas as variáveis *maxiid* e *maxballot*, pois cada um pode ter aceito diferentes instâncias com *ballots* diferentes, nelas são guardados os valores da maior instância e *ballot* aceitos, e para que uma instância seja aceita basta verificar o *ballot* da proposta atual e comparar com o *maxballot* do *acceptor*.

Para o funcionamento do código não é necessário fazer alterações no *learner* nem no *client*, pois a lógica de funcionamento continua a mesma.

3.3 Cenários de Testes de Validação

Para garantir a funcionalidade do algoritmo, é preciso assegurar, por meio de testes, diversas possibilidades de falhas recriando diversos cenários, como por exemplo, variando o número de *acceptors*, *proposers*, *learners* e clientes dentro do sistema.

Isso é necessário pois o algoritmo é usado para diversas aplicações com diferentes números de agentes, e isso não deve ser um problema para o projetista. Os testes foram reproduzidos executando comandos no terminal Linux como será mostrado nas Seções [3.3.1](#) e [3.3.2](#).

3.3.1 Mais simples

Os primeiros testes executados foram feitos utilizando o ambiente mais simples possível conforme o diagrama da Figura 5, com apenas um *proposer* e três *acceptors*, e incluindo novos elementos como forma de progressão.

Neste exemplo o *proposer* enviou uma proposta com *ballot* 10 para a instância 1, e como não há outra proposta para ser decidida, os *acceptors* confirmam a proposta e esperam

receber um valor para ser aceito. Neste caso o procedimento é bem simples, o *ballot* se mantém e após isso ocorrem as sucessivas fases 2 para as próximas instâncias.

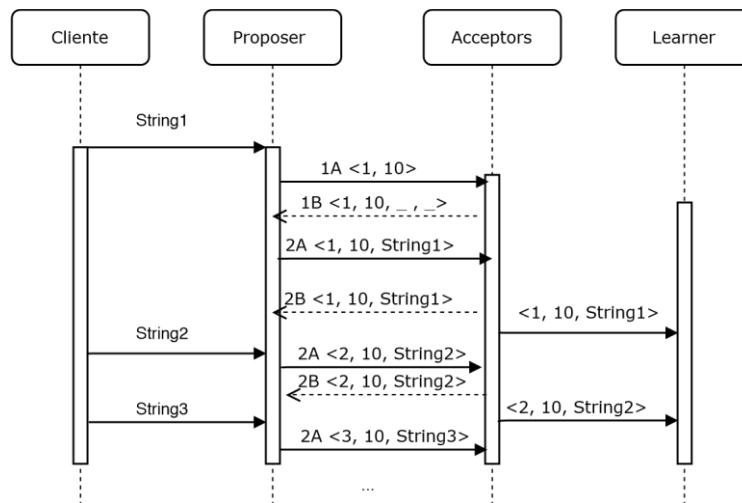


Figura 5 Ú Sequência de execução no cenário mais simples.

3.3.2 Dois ou mais proposers

Com a inclusão de mais elementos irão ocorrer interrupções por existirem diferentes propostas para a mesma instância, com isso, um novo mecanismo é utilizado. Isso é feito por uma mensagem de interrupção enviada pelos *acceptors* caso um *ballot* de outra proposta seja maior, como na Figura 6.

Neste exemplo um *proposer 0* envia uma proposta para instância 1 com *ballot* 10 e recebe uma mensagem positiva dos *acceptors*, mas em seguida uma nova proposta é enviada pelo *proposer 1* com *ballot* 11, ganhando preferência para aquela instância, isso faz com que o *proposer 0* receba uma interrupção porque 10 é menor que 11, e envia uma nova proposta com seu próximo *ballot* que neste caso seria o 20, para depois realizar a fase 2.

Caso um cliente envie um valor para o *proposer 1*, ele receberá também uma interrupção, porque o último *ballot* aceito foi o 20. Para enviar algum valor é preciso que ele realize a fase 1 com seu próximo *ballot* (neste exemplo seria o 21), e depois disso ele pode realizar a fase 2. O mesmo deve acontecer caso o *proposer 1* envie uma proposta primeiro, com *ballot* 11, se o *proposer 0* for enviar uma proposta, ele utiliza o *ballot* 20

para a instância 1. Outro exemplo seria no caso de mais de uma instância já executada, assim seria necessário realizar a fase 1 com *ballot* 20 até a instância atual.

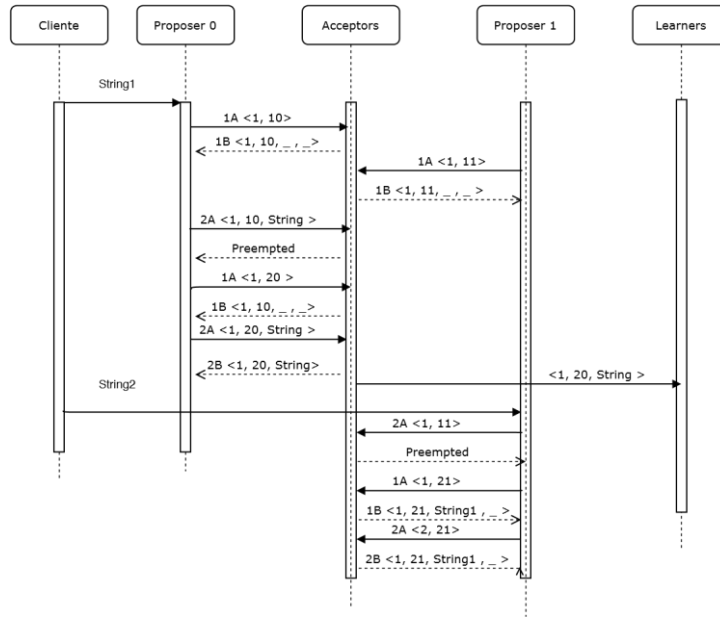


Figura 6 Ú Sequência de execução no cenário com interrupções.

Existem uma grande quantidade de cenários que podem ser criados intercalando-se essas propostas com diferentes quantidades de *proposers*, mas todas elas seguem o caso base da Figura 6, a qual novas propostas devem ser feitas utilizando um *ballot* maior do que o último recebido por algum *acceptor*.

O *ballot* é mantido para as futuras instâncias caso não haja propostas com número maior. No exemplo, caso o *proposer 1* tenha novos valores a serem propostos, as futuras instâncias, neste caso a partir da segunda em diante, serão feitas com *ballot* 21.

4 Experimentos

Para executar os testes foram utilizadas máquinas virtuais, criadas no *Amazon Elastic Compute Cloud* (Amazon EC2) que oferece uma capacidade de computação dimensionável na nuvem da *Amazon Web Services* (AWS). O serviço disponibilizado elimina a necessidade de investir em *hardware* inicialmente, portanto, podemos desenvolver e implantar aplicativos com mais rapidez.

Este serviço permite executar o número de servidores virtuais que for preciso além de possibilitar as configurações de segurança e rede desejadas, também podemos gerenciar o armazenamento individual de cada instância criada. Para utilizar este serviço é preciso fazer um cadastro antes. Depois de estar logado, o usuário inicia uma *Amazon Machine Image* para criar suas diferentes máquinas virtuais contendo qualquer aplicação desejada (AMAZON, 2015).

A imagem escolhida foi o *Linux Ubuntu* com arquitetura 64 bits x86 com *kernel* 5.3.0-1019-aws, com uma configuração de *hardware* contendo uma CPU de 2.5 GHz, memória RAM de 1 GB e armazenamento do tipo *elastic block storage* (Unidade usada pela Amazon para armazenamento de dados) que pode variar de acordo com a necessidade desejada.

Foram criadas cinco instâncias, para que cada uma representasse um agente dentro do teste, dentre elas 3 executando um *acceptor* cada, 1 executando o *learner* e o *proposer* e por fim outra para executar o *client*, internamente elas se comunicaram com a rede padrão da Amazon e foram acessadas remotamente com uma chave privada utilizando protocolo SSH.

Os principais pontos a serem observados a fim de comparar o código original com o código otimizado são relacionados com o tempo, afinal esses sistemas devem suportar a maior quantidade possível de pessoas interagindo entre si sem que a performance fique comprometida. Porém, quanto maior o número de clientes ou usuários utilizando o sistema, mais processamento deve ser feito, tornando esse ponto crucial do projeto.

Para realizar os testes foram executadas a biblioteca original e a biblioteca otimizada separadamente para que as informações fossem colhidas do log do cliente. Lá estão informações como a quantidade de valores enviados por segundo e a latência média, lembrando que os valores gerados pelo cliente são *Strings* com 64 caracteres aleatórios.

4.1 Teste de Latência

O primeiro teste foi feito para observar o tempo de espera entre o envio de uma proposta, feita por um *proposer*, e o recebimento de alguma resposta, enviada por um *acceptor*, os testes foram feitos para diferentes quantidades de *clients* já que esse poderia ser um importante parâmetro para saturar a capacidade de fazer novas propostas do *proposer*.

Para cada execução de uma certa quantidade de clientes, a biblioteca já calcula a latência média e atualiza esse valor a cada segundo, para termos um único valor para representar a latência basta pegar esses valores médios gerados e fazer novamente a média entre eles, por fim os valores foram colocados no gráfico da Figura 7.

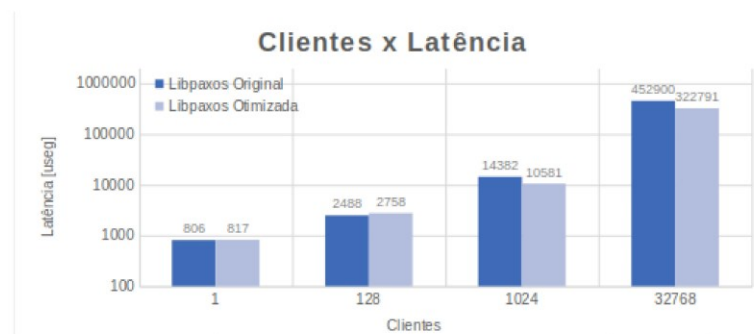


Figura 7 – Comparação da Latência relacionado com o numero de clientes.

Podemos perceber que os valores são bem próximos para poucos clientes. À medida que a quantidade de clientes aumenta, a versão otimizada apresenta latência consideravelmente menor.

4.2 Teste de Vazão

Outra variável observada foi a quantidade de valores sendo enviadas, aqui também os testes foram feitos aumentando-se o número de clientes.

Conforme o número de clientes aumenta, maior a chance de *timeouts* acontecerem, devido a quantidade de instâncias criadas. Para casos com grandes quantidades de *clients*, como por exemplo acima de 16 mil, esse problema pode acontecer logo no início da execução, sendo necessário realizar novamente o teste. Raramente acontece um problema assim para os casos com poucos clientes.

As informações sobre a quantidade de valores enviados também ficam armazenada no *log* do *client*, assim como no teste de latência. Em cada execução os dados são atualizados a cada segundo, sendo preciso fazer uma média para gerar o gráfico da Figura 8.

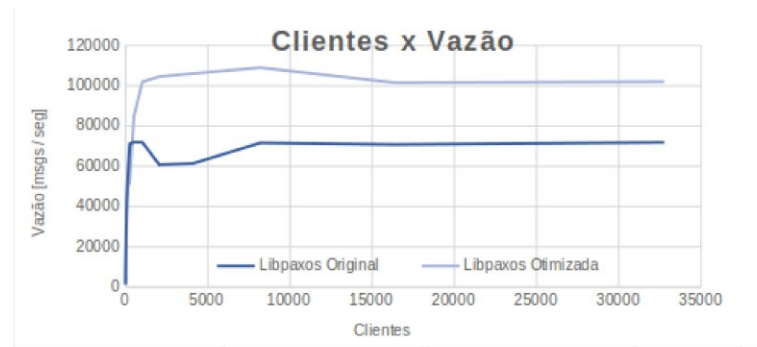


Figura 8 – Comparação da vazão relacionado com o numero de clientes.

Nesse gráfico é possível observar com clareza o grande benefício da otimização introduzida neste trabalho. Enquanto a vazão máxima da biblioteca original é de aproximadamente 70000 decisões por segundo, a versão otimizada chega a 110000 valores por segundo, representando um aumento de 60% na performance.

4.3 Comparação de latência e vazão

Por fim podemos comparar o gráfico da latência e da vazão apresentado na Figura 9.

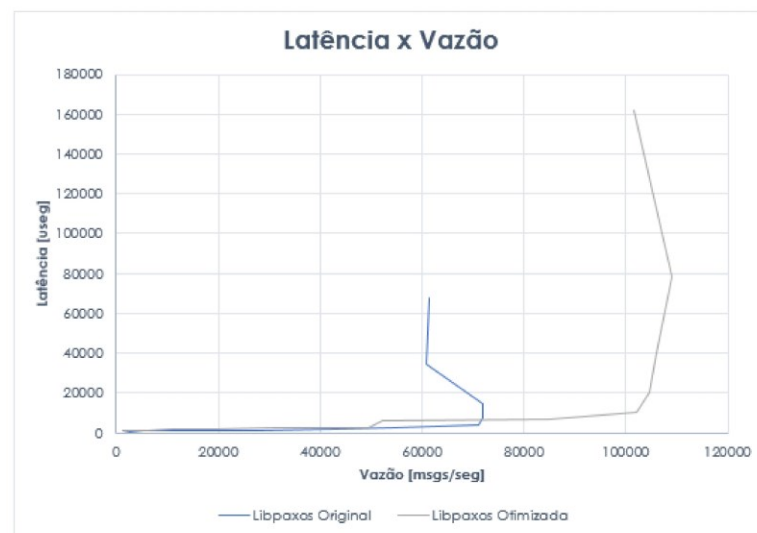


Figura 9 – Comparação entre vazão e latência.

A biblioteca original atinge cerca de 70000 valores por segundo. A partir desse ponto, a latência dispara, indicando a saturação do sistema e posterior degradação da performance a medida que mais clientes são inseridos nos testes. A versão otimizada atinge 110000 valores por segundo, e também satura a partir deste ponto, como pode ser observado na figura.

Com isso confirmamos que a biblioteca otimizada consegue fazer com que mais valores sejam enviados entre os agentes com menor atraso quando comparadas as mesmas quantidades de clientes sendo, assim, mais eficiente.

5 Conclusão

Com a implementação da proposta feita em *Stoppable paxos* (LAMPORT; MALKHI; ZHOU, 2008) na biblioteca *LibPaxos* podemos perceber, de acordo com os testes, uma melhora de desempenho, tornando seu funcionamento mais eficiente. Sabendo que na prática quanto mais valores forem decididos em um determinado intervalo de tempo, melhor para as aplicações, ainda mantendo as características de tolerância a falhas e segurança definidas no protocolo.

Essa otimização foi possível devido à execução de várias instâncias do Paxos, conhecido como *Multi-Paxos*, e permite que os usuários da biblioteca tenham uma opção otimizada para usar em suas aplicações. Podemos encontrar diversas ideias de otimização para serem aplicadas, por isso dizemos que o Paxos é uma família de protocolos. Podemos comparar quais vantagens possuem uma implementação sobre a outra.

Este trabalho trouxe uma representação de uma ideia já utilizada para dentro de uma biblioteca bastante utilizada no meio acadêmico. Isso torna possível gerar novos modelos a partir da inclusão de novos métodos, para tornar seu funcionamento ainda mais otimizado e contribuir para implementação de ideias já existentes.

Referências

- AMAZON, E. Amazon web services. Available in: <http://aws.amazon.com/ec2/>, 2015. Citado na página 25.
- BOICHAT, R. et al. Deconstructing paxos. *ACM Sigact News*, ACM New York, NY, USA, v. 34, n. 1, p. 47-67, 2003. Citado 2 vezes nas páginas 7 e 10.
- CASON, D. et al. The role of synchrony on the performance of paxos: O papel da sincronia no desempenho de paxos. [sn], 2017. Citado na página 11.
- CHANDRA, T. D.; GRIESEMER, R.; REDSTONE, J. Paxos made live: an engineering perspective. In: *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. [S.l.: s.n.], 2007. p. 398-407. Citado na página 11.
- ESPOSITO, E. G.; COELHO, P.; PEDONE, F. Kernel paxos. In: *IEEE. 2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*. [S.l.], 2018. p. 231-240. Citado 3 vezes nas páginas 6, 7 e 14.
- HOWARD, H.; MALKHI, D.; SPIEGELMAN, A. Flexible paxos: Quorum intersection revisited. In: *20th International Conference on Principles of Distributed Systems*. [S.l.: s.n.], 2017. Citado na página 7.

LAMPORT, L. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, ACM, v. 16, n. 2, p. 133-169, 1998. Citado 3 vezes nas páginas 6, 7 e 11.

LAMPORT, L. Generalized consensus and paxos. Technical Report MSR-TR-2005-33, Microsoft Research, 2005. Citado na página 6.

LAMPORT, L. Fast paxos. *Distributed Computing*, Springer, v. 19, n. 2, p. 79-103, 2006. Citado 2 vezes nas páginas 6 e 15.

LAMPORT, L.; MALKHI, D.; ZHOU, L. Stoppable paxos. *TechReport, Microsoft Research*, 2008. Citado 2 vezes nas páginas 8 e 29.

LAMPORT, L.; MASSA, M. Cheap paxos. In: IEEE. *International Conference on Dependable Systems and Networks, 2004*. [S.l.], 2004. p. 307-314. Citado na página 14.

LAMPORT, L. et al. Paxos made simple. *ACM Sigact News*, v. 32, n. 4, p. 18-25, 2001. Citado 2 vezes nas páginas 11 e 14.

LAMPORT, L. B. *Generalized paxos*. [S.l.]: Google Patents, 2010. US Patent 7,698,465. Citado na página 6.

LAMPSON, B. The abcd's of paxos. In: *PODC*. [S.l.: s.n.], 2001. v. 1, p. 13. Citado na página 10.

PRISCO, R. D.; LAMPSON, B.; LYNCH, N. Revisiting the paxos algorithm. In: SPRINGER. *International Workshop on Distributed Algorithms*. [S.l.], 1997. p. 111-125. Citado na página 6.