UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Ronistone Gonçalves dos Reis Junior

Persistência de estado em disco para o Kernel Paxos

Uberlândia, Brasil

2021

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Ronistone Gonçalves dos Reis Junior

Persistência de estado em disco para o Kernel Paxos

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

2021

Ronistone Gonçalves dos Reis Junior

Persistência de estado em disco para o Kernel Paxos

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.

Trabalho aprovado. Uberlândia, Brasil, 28 de outubro de 2021:

Paulo Rodolfo da Silva Leite Coelho
Orientador

Luis Fernando Faina

Rafael Pasquini

Uberlândia, Brasil
2021

Resumo

Este trabalho implementa o requisito de persistência em disco do estado do protocolo de *Paxos* na biblioteca *Kernel Paxos*, que busca a redução no *overhead* de comunicação, por meio de execução no *kernel space* e evitando a pilha *TCP/IP*. Como resultado obteve-se a persistência no *user space* e um processo de comunicação entre *user space* e *kernel space*. Também é apresentada uma avaliação comparativa entre as versões no *user space* antes e após a persistência em disco.

Palavras-chave: Paxos, kernel, linux, persistência

Lista de ilustrações

Figura 1 – Cenário do problema dos dois generais. 8
Figura 2 – Processo de Paxos. 14
Figura 3 – Dois *Proposers* e o mecanismo que garante que novo valor não pode ser decidido. 15
Figura 4 – Dois *Proposers* e o problema de terminação. 16
Figura 5 – Formato das mensagens do Kernel Paxos. 18
Figura 6 – Arquitetura do *Kernel Paxos*. 19
Figura 7 – Arquitetura do Processo de Persistência. 21
Figura 8 – Thread Pool de tarefas. 22
Figura 9 – Arquitetura Completa do Acceptor. 23
Figura 10 – Arquitetura dos Testes de Persistência. 24
Figura 11 – Arquitetura completa do Kernel Paxos Com Persistência. 25
Figura 12 – Experimentos com Persistência em Memória. 27
Figura 13 – Experimentos com Persistência em Disco Síncrono. 29
Figura 14 – Experimentos com Persistência em Disco Assíncrono. 30

Lista de abreviaturas e siglas

IP	Internet Protocol
LKM	Linux Kernel Module
TCP	Transmission Control Protocol

Sumário

1 INTRODUÇÃO . . . **8**
1.1 Justificativas e Objetivos . . . **10**
1.2 Método . . . **10**

2 FUNDAMENTAÇÃO TEÓRICA . . . **11**
2.1 Conceitos Básicos . . . **11**
2.1.1 Disponibilidade e Confiabilidade . . . 11
2.2 Replicação de Máquinas de Estados . . . **11**
2.3 Problema do Consenso . . . **12**
2.4 Paxos . . . **13**
2.4.1 Processo . . . 13
2.4.2 Garantia da Propriedade de Acordo . . . 14
2.4.3 Problema da Terminação . . . 15
2.5 Multi Paxos . . . **16**
2.6 Kernel Paxos . . . **17**
2.6.1 *Overhead* da Troca de Contexto . . . 17
2.6.2 Quadro Ethernet . . . 18
2.6.3 Arquitetura . . . 19

3 DESENVOLVIMENTO . . . **20**
3.1 Situação Atual do Kernel Paxos . . . **20**
3.2 Implementação da Persistência . . . **20**
3.2.1 Criação do Processo de Persistência . . . 20
3.2.2 Incorporação no Kernel Paxos . . . 22
3.2.3 Thread Pool . . . 22
3.2.4 Processamento em Lote . . . 23
3.3 Teste e Validação . . . **23**
3.3.1 Processo de Persistência . . . 23
3.3.2 Kernel Paxos com Persistência em Disco . . . 24

4 RESULTADOS . . . **26**
4.1 Análise de Persistência em Memória . . . **26**
4.2 Análise de Persistência em Disco . . . **27**
4.2.1 Análise de Persistência em Disco Síncrono . . . 29
4.2.2 Análise de Persistência em Disco Assíncrono . . . 30
4.3 Resultados - Conclusão . . . **31**

5 CONCLUSÃO 32

Referências 33

1 Introdução

Com a crescente demanda por mais poder de processamento para serviços e aplicações online, além de confiabilidade e segurança, surge a necessidade de utilizar mais sistemas computacionais que se comuniquem entre si, os quais chamamos de sistemas distribuídos. Estes sistemas trazem outras necessidades que precisam ser atendidas para um bom funcionamento, como disponibilidade, escalabilidade e transparência.

Este trabalho tem seu foco no requisito de disponibilidade. Uma maneira trivial de aumentar a disponibilidade de um serviço, evitando sua interrupção, é a adição de mais cópias deste serviço. Deste modo, pode-se tolerar a falha de algumas destas réplicas sem que o serviço pare.

Um dos grandes desafios no fornecimento de serviços replicados é a manutenção da consistência entre as réplicas, isto é, sabendo que as réplicas podem falhar e da existência de acesso concorrente às mesmas, o estado da aplicação pode divergir de uma réplica para outra, gerando respostas incoerentes para os usuários do sistema.

Uma das causas de inconsistência está no fato de se utilizar da rede para a comunicação entre os processos, que é um meio não confiável. Para exemplificar isto podemos citar o paradoxo dos dois generais [9].

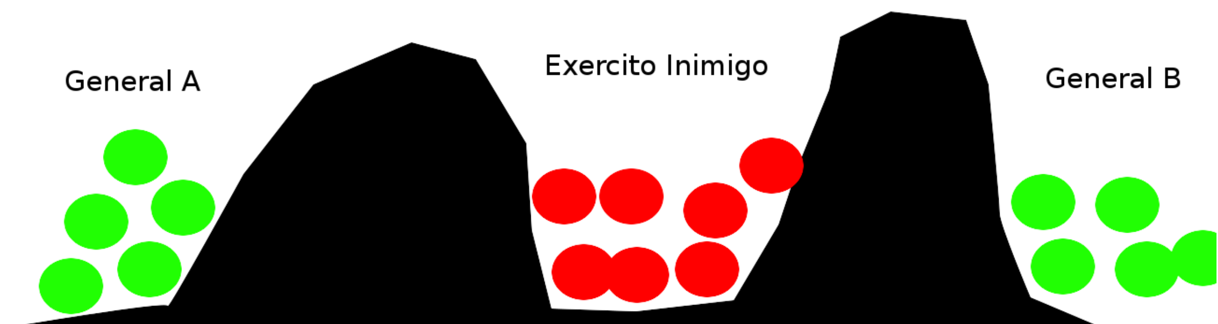


Figura 1 – Cenário do problema dos dois generais.

De forma geral, temos dois generais, que estão atrás de montanhas diferentes, com a intenção de atacar o exército inimigo em um vale entre estas montanhas (Figura 1). Para decidirem quando atacar eles precisam trocar mensagens através de um mensageiro, este para entregar a mensagem precisa atravessar o vale onde está o exército inimigo para alcançar o outro general. É na troca de mensagem que temos o problema, pois o mensageiro pode ser capturado no caminho, assim a mensagem nunca seria entregue ao outro general, o que leva o general que a enviou a hesitar em efetuar o ataque. Uma solução para isto seria esperar uma confirmação, mas quando a confirmação é enviada de

volta ao primeiro general, ela também corre o risco de ser perdida no caminho. Isto leva o segundo general a hesitar, pois ele sabe que se a confirmação não chegar ele pode atacar sozinho e seu exército ser dizimado. Esta incerteza nos leva a uma possível infinita troca de mensagens onde nunca é possível decidir o horário de ataque.

O problema dos dois generais nos mostra claramente o problema clássico que temos quando dependemos da rede para comunicação e, com sistemas distribuídos, isto não é diferente. Este tipo de falha causa inconsistência nos processos, e é aqui que surge a necessidade de uma forma de resolver ou minimizar ao máximo tal problema. Como decidir quando aceitar, por exemplo, um comando no sistema? Como garantir que a resposta desse sistema será a mesma para aqueles que o utilizar?

Nesta direção, temos um problema clássico nomeado de problema do consenso [2]. O problema consiste em um grupo de processos que possuem valores a serem propostos e devem decidir o mesmo valor entre os propostos. Precisamos de uma forma determinística de gerar consenso entre os processos do sistema, para que possam decidir quando aceitam um comando e o executam, alterando assim o seu estado.

Paxos [7] é um algoritmo que procura resolver o problema do consenso. O algoritmo trabalha em duas fases e com três papeis: *proposers*, *acceptors* e *learners*. Os *proposers* que geram comandos para serem aceitos pelos *acceptors*, que por sua vez votam em propostas, que uma vez aceitas são enviadas para os *learners* executarem comandos.

O Algoritmo de Paxos nos dá a possibilidade de criar uma replicação de máquinas de estado, que é uma forma simples de replicação de processos, ou seja, um grupo de processos que parte do mesmo estado inicial e executa a mesma sequência de comandos de modo determinístico e nos permite criar sistemas tolerantes a falhas. Pelo fato do Paxos necessitar da maioria dos processos para atender o funcionamento, temos então uma tolerância a falhas, em um sistema com n processos, de $\lfloor \frac{n-1}{2} \rfloor$ processos.

Existem várias implementações de Paxos que se beneficiam de cenários específicos para atingirem alta performance. Como por exemplo topologias de rede [5, 13], hardware especializado [10, 3] e de peculiaridades do problema a ser tratado [11, 14, 8].

O Kernel Paxos [4] propõe uma implementação que não depende de nenhum destes três cenários, oferecendo uma implementação mais genérica e ótima para todos os cenários.

Baseando-se na *libpaxos* [4], uma implementação open source, o Kernel Paxos traz o algoritmo de Paxos para o kernel linux através de um *LKM*, ou *Linux Kernel Module*, visando uma otimização em relação a troca de contexto entre o *user-space* e o *kernel-space*. Além disso realiza uma otimização na troca de mensagens, ou seja, ao invés de utilizar a pilha TCP/IP, ele utiliza quadros Ethernet diretamente, trazendo assim uma otimização em relação a comunicação do protocolo.

Para permitir a recuperação das réplicas em caso de falha, o estado da aplicação

deve ser persistido em disco. Este trabalho foca na implementação da persistência que é necessária para o funcionamento correto do algoritmo em caso de uma recuperação.

1.1 Justificativas e Objetivos

Para que aplicações que utilizam o algoritmo de Paxos possam tolerar falhas e permitir a recuperação do estado da aplicação, os *acceptors* devem persistir em disco cada mensagem que será enviada aos *proposers* [7]. Uma deficiência do Kernel Paxos é a inexistência de tal persistência.

O presente trabalho visa desenvolver e implementar um mecanismo que permita persistir informações de módulos do kernel do Linux em disco. Uma vez implementado e validado, tal mecanismo será integrado ao Kernel Paxos para suprir a deficiência apresentada e permitir a recuperação do estado dos *acceptors* a partir do disco.

1.2 Método

Neste trabalho está sendo estudado formas de persistência em disco a partir do Kernel, visto que o nosso primeiro desafio é que, a persistência realizada diretamente do Kernel no disco não é recomendada por desenvolvedores do Kernel Linux.

A primeira etapa (tarefa 1) consiste em elencar as possíveis formas de persistência a partir do Kernel do Linux. Tais formas foram implementadas e os desempenhos avaliados e comparados (tarefa 2). O modelo que apresentou melhor performance foi incorporado aos *acceptors* do Kernel Paxos para permitir salvar seus estados em disco (tarefa 3). O passo seguinte implica na implementação de mecanismo de recuperação dos *acceptors* após uma falha (tarefa 4). Finalmente, a biblioteca foi comparada com os modelos de persistência em memória e em disco para determinar o impacto no desempenho geral do sistema (tarefa 5). Cada tarefa tem duração aproximada de um mês. O último mês é dedicado à preparação do documento final.

2 Fundamentação Teórica

Neste capítulo são apresentados os conceitos básicos e necessários para compreensão deste trabalho bem como os trabalhos relacionados.

2.1 Conceitos Básicos

Neste capítulo trabalharemos com um conjunto de instâncias $\{s_1, s_2, s_3, ..., s_n\}$, onde juntas formam um sistema $S = \{s_1, s_2, s_3, ..., s_n\}$. Estas instâncias são replicas de um serviço que buscaremos a confiabilidade e tolerância a falhas. O número n é fixo e conhecido.

2.1.1 Disponibilidade e Confiabilidade

Podemos ver a disponibilidade como o tempo onde o sistema está disponível, podendo fazer o cálculo da porcentagem de disponibilidade através da equação $\frac{Tempo\ disponivel}{Tempo\ esperado} \times 100\%$

E a confiabilidade é vista como a probabilidade do funcionamento correto, ou seja, o sistema precisa estar disponível e respondendo corretamente, uma resposta errônea impacta a confiabilidade de um sistema.

2.2 Replicação de Máquinas de Estados

Quando chegamos ao ponto de distribuir nossos sistemas, há a necessidade de manter a consistência e a confiabilidade entre as nossas réplicas, ou seja, este sistema precisa que cada replica tenha os mesmos dados e que a resposta para uma operação executada em qualquer uma das réplicas seja igual. Uma solução para tal problema é fazer com que uma réplica possa ser construída de forma determinística, e se tratando deste cenário uma máquina de estados nos fornece este ganho.

Uma Máquina de Estados é definida por um conjunto de estados e operações, onde cada operação deve ser[15]:

A.1 Determinística;

A.2 Atômica em relação às outras operações.

Devido à **A.1** temos garantia de que sempre que nosso sistema estiver em um estado α e uma operação λ for aplicada teremos um estado β, independente de quantas vezes este cenário ocorra, sempre teremos o estado β como resultado.

Dado a definição de Máquina de Estados, temos garantias de que, nossas instâncias não falhas tenham o mesmo estado e forneçam a mesma resposta para seus clientes. Nos trazendo agora o problema de como replicar as operações para todas as instâncias, de forma que todas as instâncias não falhas executem as operações na mesma ordem, permitindo ao cliente ter o mesmo resultado independente da instância que o atenda.

Uma solução simples é termos um único líder, este será responsável por executar todas as operações, replicando para seus seguidores de forma a deixá-los sincronizados. Esta solução permite que em caso de falha do líder um seguidor assuma a liderança devido à sincronia de máquinas de estados, fornecendo ao sistema uma tolerância a falhas. Esta solução traz algumas complexidades e problemas a serem trabalhados, como, a escolha de um líder dentre todas as instâncias, garantia de que, em caso de uma instância falha trabalhe como um segundo líder, as outras instâncias não falhas não executem operações originadas por este líder falho.

Frente a estes problemas, há a necessidade de que o sistema utilize um protocolo que forneça garantias de que uma instância falha não comprometa completamente a consistência do nosso sistema.

2.3 Problema do Consenso

Na replicação de máquinas de estado há a necessidade de garantir um acordo entre as réplicas, onde de alguma forma tenhamos a garantia que as réplicas vão decidir a mesma sequência de operações. Este problema pode ser visto como um Problema do Consenso, que consiste em como garantir que dentre diferentes operações propostas, haja unanimidade na escolha. O Problema do consenso pode ser definido pelas seguintes propriedades: [4]:

1. Integridade: Para que um valor λ seja decidido, o mesmo precisa ter sido proposto por alguma instância
2. Termino: Toda instância não falha decide por algum valor
3. Acordo: Caso uma instância se decida pelo valor λ então nenhum processo não falho se decide por outro valor

O Problema do Consenso também pode ser visto em processos como eleição de líderes, onde desejamos garantir que teremos apenas um líder, ou até mesmo no *commit* de um banco de dados distribuído, onde é desejado uma consistência em seus nós e a tolerância à falha.

2.4 Paxos

O Paxos [7] é um protocolo de consenso tolerante à falha, que utiliza três papéis: *Proposer*, *Acceptor*, *Learner*. Trabalhando em duas fases, os papéis procuram uma execução de consenso em um meio em que instâncias podem falhar e mensagens podem ser perdidas, de forma que o *Proposer* tem o objetivo de propor operações para o *Acceptor*, que visa garantir o consenso, para enviar ao *Learner* o valor que deve ser aprendido.

2.4.1 Processo

Quando temos apenas um *Acceptor*, uma forma simples de termos uma decisão é de apenas aceitar a primeira proposta que receber, mas quando temos vários *Acceptors* e vários *Proposers*, diferentes propostas podem chegar em *Acceptors* diferentes. Para resolver isto, podemos definir um quorum mínimo para que uma proposta seja aceita, ou seja, quantos *Acceptors* precisam aceitar uma proposta para que posssa ser considerada aprendida. Para garantir que apenas uma proposta será aceita, devemos definir que ela precisa ser aceita pela maioria dos *Acceptors*. Este fato nos dá também o número de falhas que o sistema é tolerante, pois para termos maioria em um conjunto com n *Acceptors* precisamos de $\lfloor \frac{n}{2} \rfloor + 1$, permitindo assim uma falha de $\lfloor \frac{n-1}{2} \rfloor$.

A Comunicação entre o *Proposer* e o *Acceptor* é dado em duas fases e duas mensagens em cada fase, que podemos nomear de 1A, 1B, 2A e 2B, todo o processo abaixo é ilustrado na Figura 2:

Fase 1. O *Proposer* escolhe um número único de forma sequencial denominado rodada para aquela proposta e envia a mensagem 1A para o *Acceptor*, que deve responder apenas se não conhecer proposta com valor superior à rodada desta proposta, enviando a mensagem 1B, que é uma promessa de não aceitar nenhuma proposta com rodada menor do que a recebida. Em caso de uma falha nesse processo, seja por já existir proposta maior do que a sugerida pelo *Proposer*, por não atingir o quorum necessário ou por alguma falha na comunicação, o *Proposer* precisa tentar um novo valor de rodada superior ao anterior, executando novamente esta fase.

Fase 2. Após o *Proposer* receber a promessa de aceitação(1B) de um quorum de *Acceptors*, ele envia a mensagem 2A, que possui o número da proposta e o valor a ser proposto. Ao receber a mensagem 2A, o *Acceptor* responde apenas caso não tenha recebido algum valor n, maior do que o recebido entre o tempo de envio da 1B e a 2A, confirmando ser o maior valor proposto ele pode aceitar a proposta. Uma forma de prosseguir é o *Acceptor* enviar para os *Learners* e *Proposers* a confirmação de aceitação da proposta(2B). Desta forma o *Proposer* pode confirmar a aceitação da proposta e o *Learner* pode aprender o valor proposto, caso receba a mensagem 2B de um quorum de *Acceptors*.

Nas duas fases a não respostas das mensagens A (1A e 2A) com as mensagens B (1B e 2B), é identificada como uma não aceitação por parte dos *Acceptors*, sendo assim o *Proposer* precisa iniciar uma nova rodada.

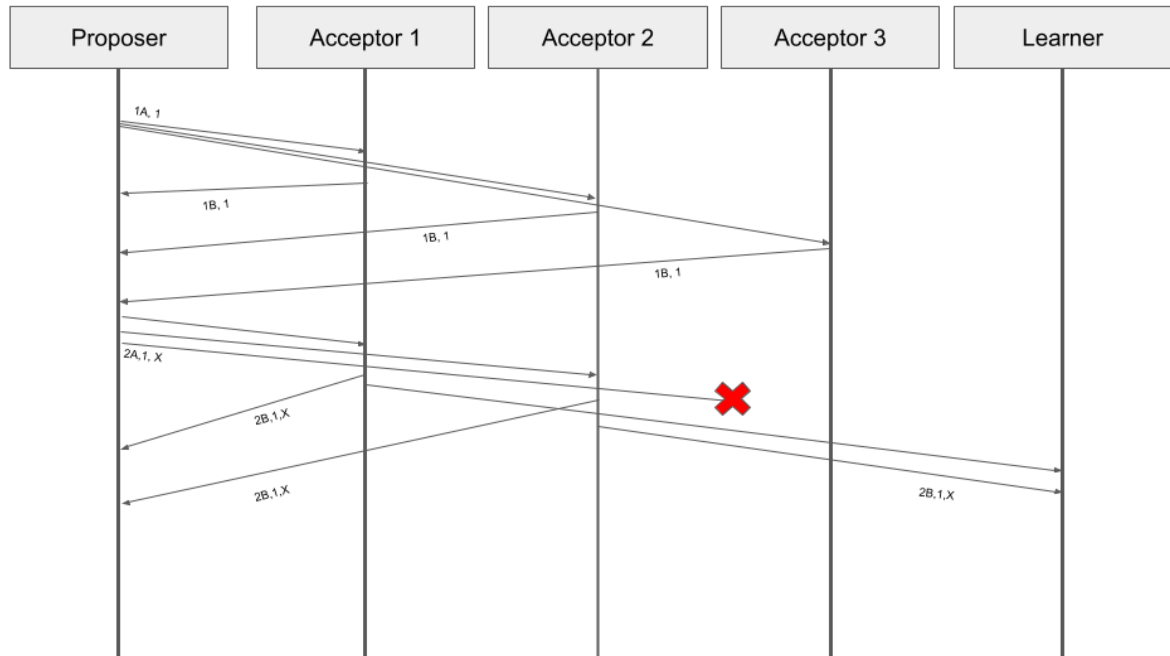


Figura 2 – Processo de Paxos.

Todo o processo descrito acima pode ser exemplificado pela Figura 2, nela podemos ver a troca de mensagens 1A, 1B, 2A e 2B, além de uma falha na comunicação no envio da 2A para o *Acceptor* 3, tal falha é tolerada, pois ao termos 3 *Acceptors* o nosso quorum precisa ser de pelo menos 2. Sendo assim o valor proposto pelo *Proposer* é aprendido pelo *Learner*.

2.4.2 Garantia da Propriedade de Acordo

Conforme visto na propriedade de 3 (Acordo), em uma rodada de Paxos após ser decido um valor, outro valor não pode ser decido, mesmo em propostas com valor de rodada superior. Para isto na mensagem 1B, ao enviar para o *Proposer* a promessa de aceitação, o *Acceptor*, caso já tenha se participado da fase 2 anteriormente, envia tal valor permitindo que o *Proposer* envie a mensagem 2A com o valor potencialmente já decidido escolhendo o maior valor de rodada conhecido[4], conforme mostra a Figura 3.

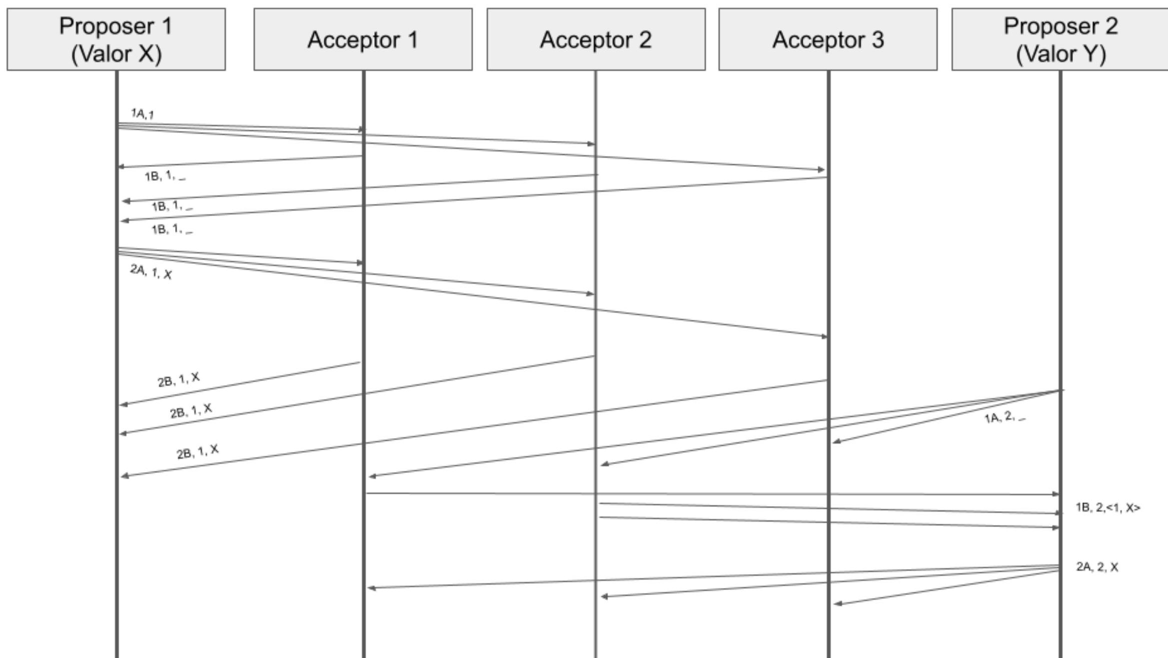


Figura 3 – Dois *Proposers* e o mecanismo que garante que novo valor não pode ser decidido.

2.4.3 Problema da Terminação

No Protocolo de Paxos há um cenário em que nunca existe um consenso[7], ou seja, uma não terminação violando a propriedade 2 *(Termino)*, este cenário ocorre quando temos uma concorrência entre múltiplos *Proposers*. Como cada *Proposer* busca a aceitação de sua proposta, pode ocorrer o seguinte cenário (descrito na Figura 4):

A existência de dois *Proposers P1* e *P2*, propondo valores *X* e *Y*, respectivamente. Após a execução da fase 1 com o número da proposta *n1* e antes do início da fase 2 pelo *P1*, o *Proposer P2* inicia a fase 1 com um número de proposta *m1*, onde $m1 > n1$, fazendo com que os *Acceptors* não aceitem *n1* durante a execução da fase 2 pelo *Proposer P1*. Desta forma para conseguir a aceitação de *X* o *Proposer P1* inicia o processo com o número de proposta *n2*, sendo $n2 > m1$, causando novamente a não aceitação na fase 2, agora executada pelo *Proposer P2*. Este processo inicia uma concorrência entre os dois *Proposers*, fazendo com que não seja possível a finalização da fase 2, e consequentemente a não terminação da instância do protocolo.

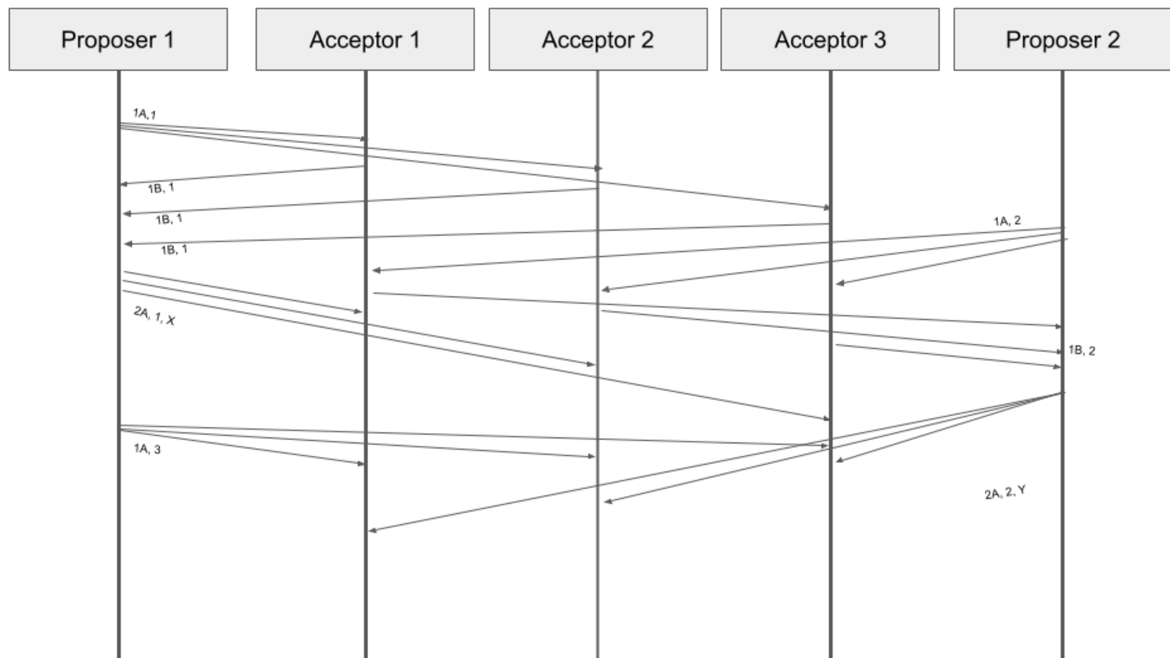


Figura 4 – Dois *Proposers* e o problema de terminação.

Para garantir a Terminação do protocolo de Paxos, um *Proposer* deve ser eleito como um líder, este *Proposer* será o único que poderá realizar propostas aos *Acceptors* de forma que não haja concorrência entre os demais *Proposers*.

2.5 Multi Paxos

Uma decisão tomada pelo algoritmo de Paxos não pode ser mudada, ou seja, depois de uma operação ser aprendida não há como alterar. Frente ao problema de implementar uma máquina de estados, onde cada operação altera o estado é preciso realizar várias operações, este fato é problemático. Para resolvermos isto, são utilizadas várias execuções de Paxos. Sempre que é necessário a execução de uma operação, é executada uma instância de Paxos para garantir o consenso e a consistência em todas as instâncias não falhas. Cada instância tem um número único e as réplicas executam as operações em ordem crescente do número de instância.

Desta forma, cada estado da máquina de estados é a i^a instância de Paxos, garantindo a ordem de cada valor em cada máquina de estados e fazendo com que o sistema apenas aprenda um valor caso todos os anteriores foram aprendidos.

Como explicado no Problema da terminação 2.4.3, há a necessidade de eleger um líder para o sistema. Com esta eleição de um líder há a possibilidade de otimização no processo que é pré-executar a Fase 1 de Paxos para algumas instâncias do Protocolo. Esta otimização reduz a quantidade de mensagens trocadas para aceitar um valor. Porém, existe o problema em que podem ocorrer “buracos” na máquina de estados, onde, por

exemplo, as instâncias 1-10 foram aprendidas e o líder enviou os valores 11 e 12, mas as mensagens da instância 11 falha e a instância 12 é aprendido antes de o valor da instância de Paxos 11 ser escolhido. Esse não deveria ser um problema, já que bastaria o líder retransmitir a mensagem e garantir a consistência. A falha ocorre quando o sistema está com este buraco e o líder falha, perdendo o valor que deveria ser proposto. Para sair deste cenário bastaria que o próximo líder eleito, ao identificar o buraco, transmita um valor de *noop* que irá inserir um valor apenas para preencher a falha[7].

A operação de *noop* pode ser executada porque para existir um buraco o valor não pode ter sido decidido, o que torna o sistema consistente, onde a operação de *noop* não sobrescreve outro valor, e fornece um conjunto 1-12 de valores na máquina de estados. A resolução do buraco existente nas instâncias de Paxos permite a continuação do sistema.

O Multi Paxos permite que todo histórico seja recuperado em um caso de reinício de uma máquina, pois ao executar uma instância de Paxos o valor pode ser persistido de forma a armazenar o estado atual da máquina de estados.

2.6 Kernel Paxos

O Kernel Paxos é uma implementação do protocolo de Paxos no *Kernel Linux*. Baseado no *libpaxos*, ele foca o ganho de performance removendo o tempo gasto em troca de contexto ao se utilizar recursos do *Kernel* e utilizando um quadro *Ethernet* puro.

2.6.1 *Overhead* da Troca de Contexto

O Linux é separado em dois contextos, o *User Space* e o *Kernel Space*, separação que visa proteger o *hardware* de um código com comportamento errôneo, mesmo que não intencional[4]. Toda aplicação no *User Space* que, por exemplo, precise enviar algo pela rede, irá realizar uma *System Call* ou *syscall*, que é uma forma do *User Space* realizar chamadas ao *Kernel Space* e utilizar suas funcionalidades.

Frente a isto, uma aplicação no *User Space* precisa necessariamente, ao utilizar algo do *Kernel Space*, realizar duas trocas de contexto, uma sendo do *User Space* para o *Kernel Space* e outra na resposta vinda do *Kernel Space* para o *User Space*. Isto aumenta o tempo gasto e pode ser muito significativo para aplicações do tipo *I/O Bound* (como mostrado pelo *Kernel Paxos* [4]), ou seja, que faz uso intensivo de entrada e saída, que é o caso do Protocolo de Paxos que envia e recebe diversas mensagens através da *Network Interface Card(NIC)*.

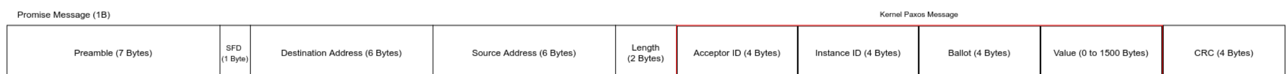
2.6.2 Quadro Ethernet

Quando o Sistema está em uma rede local(LAN), é possível fazer algumas otimizações assumindo que uma LAN é um ambiente controlado [4], como trabalhar totalmente na camada de enlace removendo o *overhead* da pilha *TCP/IP* e reduzindo a quantidade de dados trafegados. E devido ao quorum mínimo e *timeouts* para perdas de mensagens do protocolo de *Paxos* a confiabilidade das mensagens é tratado pelo protocolo, suprindo desta forma a falta do protocolo *TCP*.

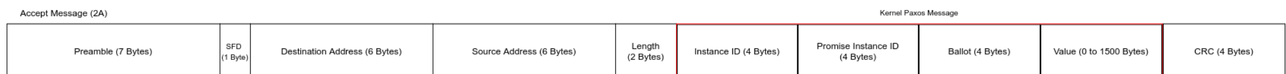
O *Kernel Paxos* utiliza a remoção das informações como cabeçalhos *IP* e *UDP*, inserindo o *Payload* utilizado pelo Protocolo diretamente no quadro *Ethernet* utilizando um tipo customizado do quadro *Ethernet*.

Essa remoção significa cerca de 40 *bytes* para o *TCP* ou 28 *bytes* para o *UDP*, sendo 20 *bytes* do protocolo IP na camada de rede, 20 *bytes* do protocolo TCP ou 8 *bytes* do protocolo *UDP* na camada de transporte. Esse *overhead* significa um aumento de 103%, no *TCP*, ou 84% no *UDP*, do tamanho da mensagem do protocolo *Kernel Paxos*. Como pode ser visto na Figura 5d que é a maior mensagem do protocolo possuindo 38 *bytes*, somado 26 *bytes* do *frame ethernet* para os *headers*, a maior mensagem do protocolo tem 64 *bytes* de *headers*.

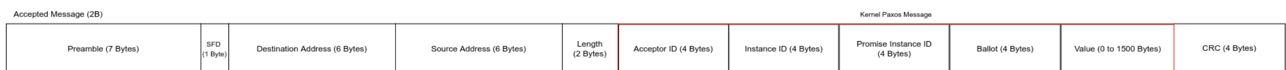
(a) Mensagem 1A Proposta de Número de Rodada.



(b) Mensagem 1B Promessa de aceitação de Número de Rodada.



(c) Mensagem 2A Proposta de Valor da Rodada.



(d) Mensagem 2B Aceitação de Valor da Rodada.

Figura 5 – Formato das mensagens do Kernel Paxos.

Para a utilização do protocolo em uma rede com múltiplas *LANs* é necessário a utilização do protocolo *TCP* ou *UDP* perdendo desta forma a otimização realizada pelo *Kernel Paxos*.

2.6.3 Arquitetura

O *Kernel Paxos* implementa todas os três papeis, que podem ser utilizados de forma separada, tendo máquinas separadas para cada papel, ou utilizar todos os papeis em uma única máquina (réplica).

Como visto na Figura 6, toda comunicação ocorre utilizando a rede, sendo apenas a comunicação entre o *Learner* e a Aplicação, que irá executar os valores aprendidos, utilizando o *Char Device*. Na Figura 6 fica claro que todas os papeis são executados no *Kernel Space*, o que exemplifica a redução do número de trocas de contextos realizados pelo protocolo, discutido na Subseção 2.6.1.

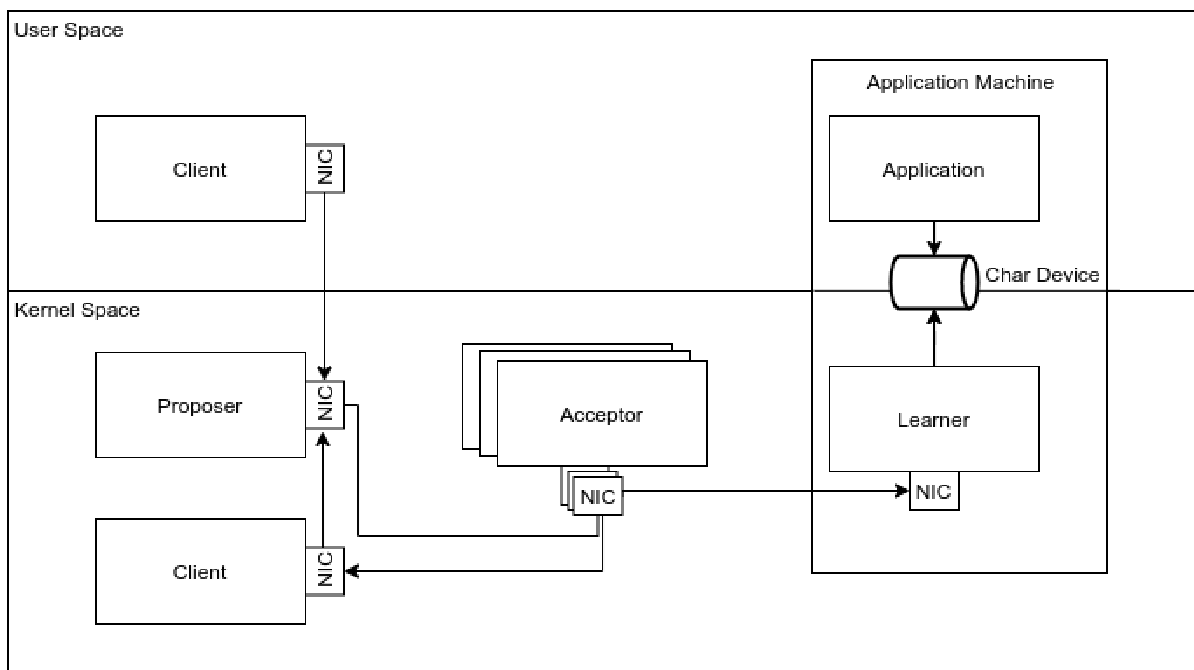


Figura 6 – Arquitetura do *Kernel Paxos*.

3 Desenvolvimento

Este trabalho fez a implementação da persistência em disco do Protocolo de Paxos, implementado pelo Kernel Paxos, de forma que uma instância tenha seu estado armazenado e recuperado caso ocorra um reinício.

3.1 Situação Atual do Kernel Paxos

Conforme visto na Seção 2.6, o Kernel Paxos aproveitou a lógica da implementação de Paxos feita pelo *libpaxos*, mas os *Acceptors* trabalhavam utilizando apenas dados salvos na memória.

Dado este fato, no Kernel Paxos ao ter a máquina reinicializada todas as execuções de Paxos feitas são perdidas. Na implementação de persistência em memória do Kernel Paxos é utilizado um *array* circular onde para acessar uma posição neste *array* é utilizado o resto da divisão do *instance id* pelo número de posições deste *array*, o *instance id* é o id definido pelo *Proposer* ao iniciar a fase 1 de uma execução de Paxos. Este *array* nos limita a um número de informações salvas, fazendo com que ao se ter um grande número de informações, novos dados sobrescrevam os antigos, causando perda do histórico do algoritmo.

3.2 Implementação da Persistência

No início do desenvolvimento foi identificado que a persistência feita diretamente do *kernel* não é uma boa prática [6] devido a problemas com segurança e a complexidade do desenvolvimento, além da necessidade de gerenciar os dados e a forma em que seriam escritos, fazer a leitura dos mesmos e garantir a consistência da base de dados. Assim, utilizar um banco de dados já implementado foi uma alternativa que fornecia segurança nos dados escritos e lidos, além de reduzir a complexidade da implementação.

Foi escolhido o *lmdb* como o banco de dados. Ele é um banco transacional chave-valor de alta performance. O *lmdb* também é utilizado pelo *libpaxos* [4].

3.2.1 Criação do Processo de Persistência

Com o banco escolhido, deve-se definir como comunicar com este banco de dados. Para isso foi escolhido o *char device*, como uma forma de enviar dados do *Kernel Space* para o *User Space*. O *char device* é um *driver* em que são implementadas no *Kernel Space* algumas *syscalls* como, *open, close, poll*(bastante utilizado neste trabalho), *read* e *write*,

que fornecem uma interface próxima à interface de um arquivo. Ao executar tais *syscalls* é executada a implementação no *Kernel Space* enviando ou recebendo *bytes*, permitindo uma comunicação entre processos em contextos diferentes[12]. Com isto em mente, foi decidido implementar o processo que comunicaria com o banco no *User Space*, e o código que manipularia o *char device* no *Kernel Space*, inicialmente separado do Kernel Paxos. Isso permitiria fazer testes mais rápidos, testando apenas o processo de leitura e escrita no banco de dados.

A arquitetura pensada (Figura 7) consiste em colocar no *User Space* um processo que trataria operações de escrita e leitura. Para isso foram utilizadas duas *threads* para que os dois tipos de operações pudessem ser tratados em paralelo. O *User Storage*, nome dado ao processo no *User Space*, faria um *polling*, ou seja, estaria sempre consultando o *char device* em busca de novas operações para executar, e iria até o *lmdb* para executar cada operação, seja leitura ou escrita. Já no *Kernel Space* existiriam dois componentes que receberiam as operações, vindas do próprio *Kernel Space*, enfileirando-as em um *buffer* para serem futuramente lidas pelo *User Storage* no processo de *polling*.

Sendo esta arquitetura a mais simples de ser implementada, utilizando *syscalls* para comunicação e um banco de dados no *user space*, o que evita, por exemplo, uma implementação que utilize a interface de rede para comunicação entre processos, exigindo todo um *overhead* de comunicação. Desta forma não foram implementadas outras alternativas como sugerido na Seção 1.2 na primeira etapa do trabalho.

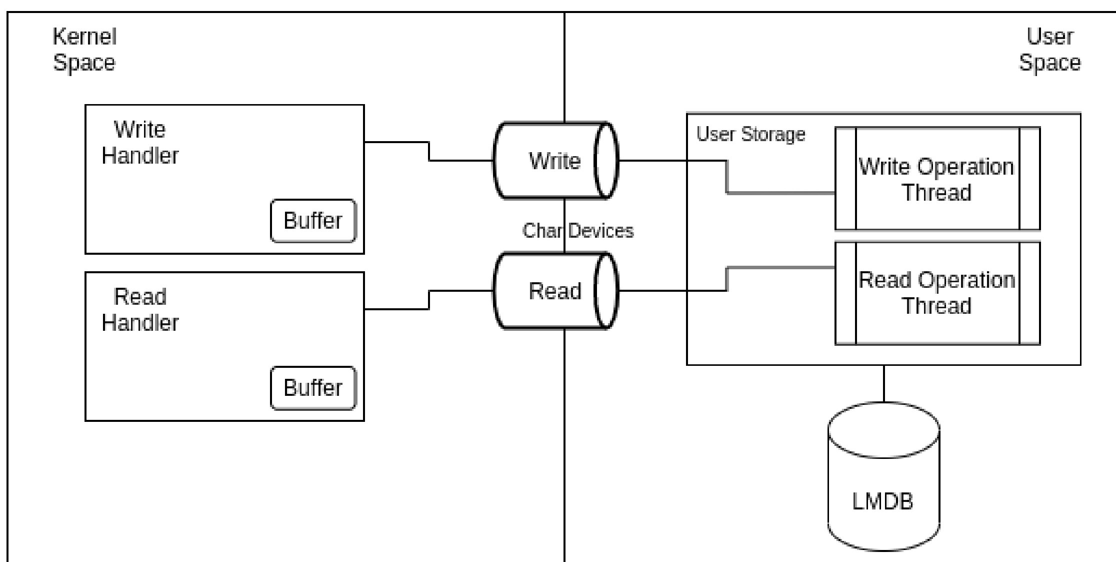


Figura 7 – Arquitetura do Processo de Persistência.

3.2.2 Incorporação no Kernel Paxos

Após todo o processo de persistência implementado, era necessário levá-lo para dentro do Kernel Paxos. O Kernel Paxos já utilizava um *char device*, mas neste caso era utilizado para enviar informações para serem lidas por uma aplicação *Learner*, que utiliza o Kernel Paxos para decidir valores. Por esse motivo, foi necessário criar outros *char devices* com responsabilidades únicas, evitando a complexidade e problemas ao trafegar mensagens de operações diferentes no mesmo meio, além de paralelizar o processamento dessas mensagens. Para cumprir esse requisito foi preciso refatorar o código que já existia do *char device* para permitir criar vários *char devices* de uma forma simples.

3.2.3 Thread Pool

Com o processo de persistência pronto para se integrar ao Kernel Paxos, existia um problema em que, quando enfileirada a operação para o *User Space*, o processo no *Kernel Space* precisaria aguardar a resposta do banco de dados para dar resposta a quem solicitou a operação na base de dados. Por exemplo, no caso de operações de leitura em que o processo que solicitou a leitura requer os dados lidos para continuar. E isso causa um "congelamento" do processo no *Kernel* causando problemas, por se tratar do *Kernel* do sistema operacional. Para resolver o problema, foi criado um *thread pool* no qual há uma lista de tarefas que são criadas pelas mensagens vindas da rede para serem processadas pelos *workers* do *thread pool* (Figura 8).

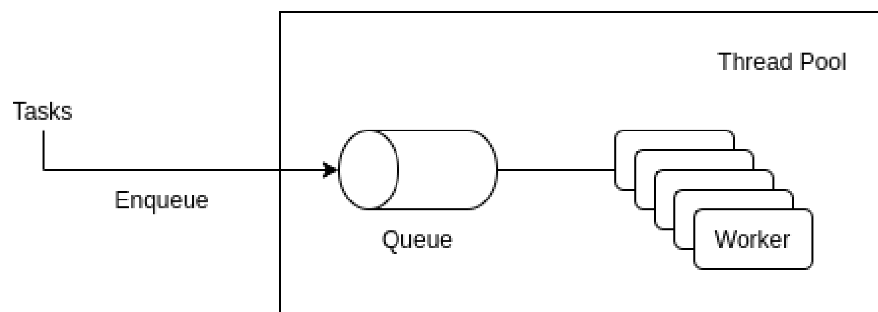


Figura 8 – Thread Pool de tarefas.

Chegando ao fim com a arquitetura da Figura 9, onde cada mensagem vinda da rede passa pelo nosso *thread pool*, que ao conseguir um *worker* disponível executa o protocolo de Paxos e quando necessário adiciona alguma operação aos *handlers* de leitura e escrita. Ao chegar no *buffer* as operações são lidas pelo *User Storage* que executa a operação no *lmdb*, retornando ao fim a resposta para o *char device*.

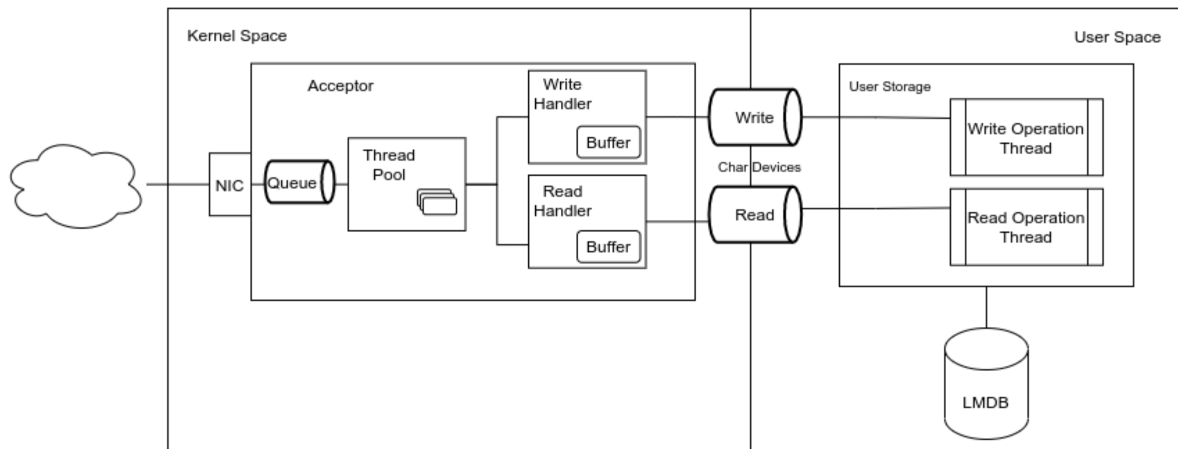


Figura 9 – Arquitetura Completa do Acceptor.

3.2.4 Processamento em Lote

Com toda a arquitetura pronta e implementada, foi possível identificar uma melhoria na busca de operações para se executar no *User Space*. A melhoria consistia em fazer as operações em lotes. Por exemplo, a *thread* de leitura poderia buscar um lote de tamanho 100 de operações no *char device*, respondendo também as 100 operações em uma única escrita no *char device* para o *Kernel Space*. Essa otimização reduz o *overhead* drasticamente que temos ao ir até o *Kernel Space* e voltar.

3.3 Teste e Validação

3.3.1 Processo de Persistência

Todo o processo realizado na Seção 3.2.1 utiliza apenas um processo no *Kernel Space*. No *User Space* existiam dois processos: O *User Storage*, responsável pelas operações no *lmdb*, e um processo de teste que simulava solicitações do protocolo de Paxos ao seu processo de persistência.

O processo de testes realizou o envio de mensagens para um *char device* de testes que simulava uma mensagem vinda da rede tanto para leitura quanto para escrita, sendo dois *char devices*, um para cada operação. No *Linux Kernel Module (LKM)* o processo que trata as operações do *char device* envia a mensagem para serem tratadas direto para o *buffer* do processo que executa as operações na persistência. O processo de persistência possuí também dois *char devices* para comunicação com o *User Storage*. Utilizando este processo foi possível validar a integridade dos dados enviados e recebidos da persistência, tal como o bom o funcionamento de todo o processo de persistência, antes de ser incorporado no Kernel Paxos (ilustrado na Figura 10). Neste momento não foram feitos testes de performance do processo isolado do *Kernel Paxos*.

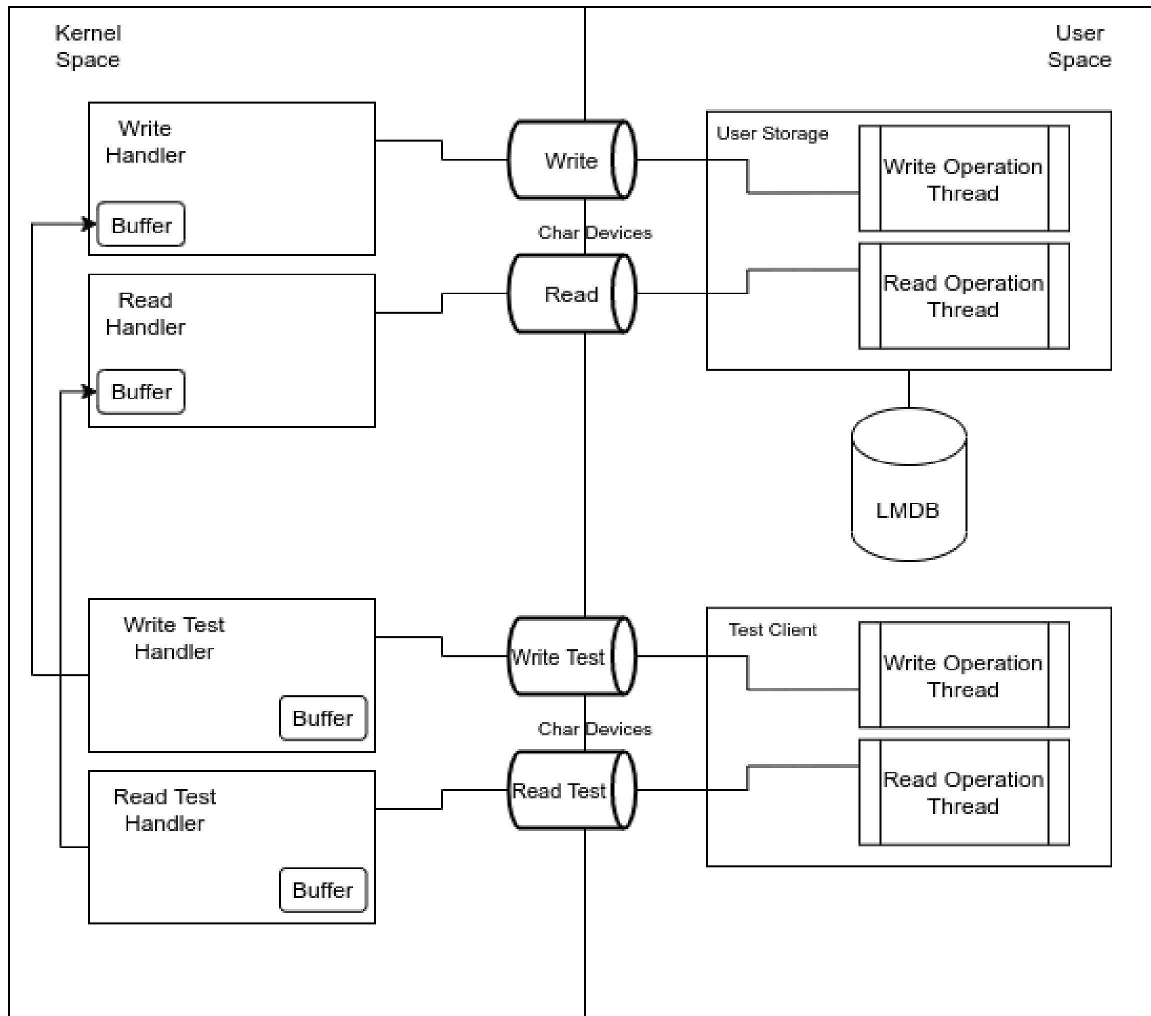


Figura 10 – Arquitetura dos Testes de Persistência.

3.3.2 Kernel Paxos com Persistência em Disco

Já no Kernel Paxos, a única mudança em relação ao Kernel Paxos original (Figura 6) foi a adição do *User Storage* com *lmdb* e os *char devices*, como ilustrado pela Figura 11. Todos os testes realizados utilizaram um *client* para gerar dados para serem decididos pelo protocolo. O *client* é também um *learner* de forma que é possível medir o tempo entre a proposta e a aceitação do valor, além de calcular a quantidade de mensagens por segundo que o protocolo consegue dar vazão.

Para validar o correto funcionamento do processo de persistência no Kernel Paxos foi validado se um valor recebido sofreu alterações após ser proposto. Desta forma é possível confirmar que todos os dados salvos na persistência estão corretos, pois toda a leitura e escrita feita pelos *Acceptors* utiliza o processo de persistência.

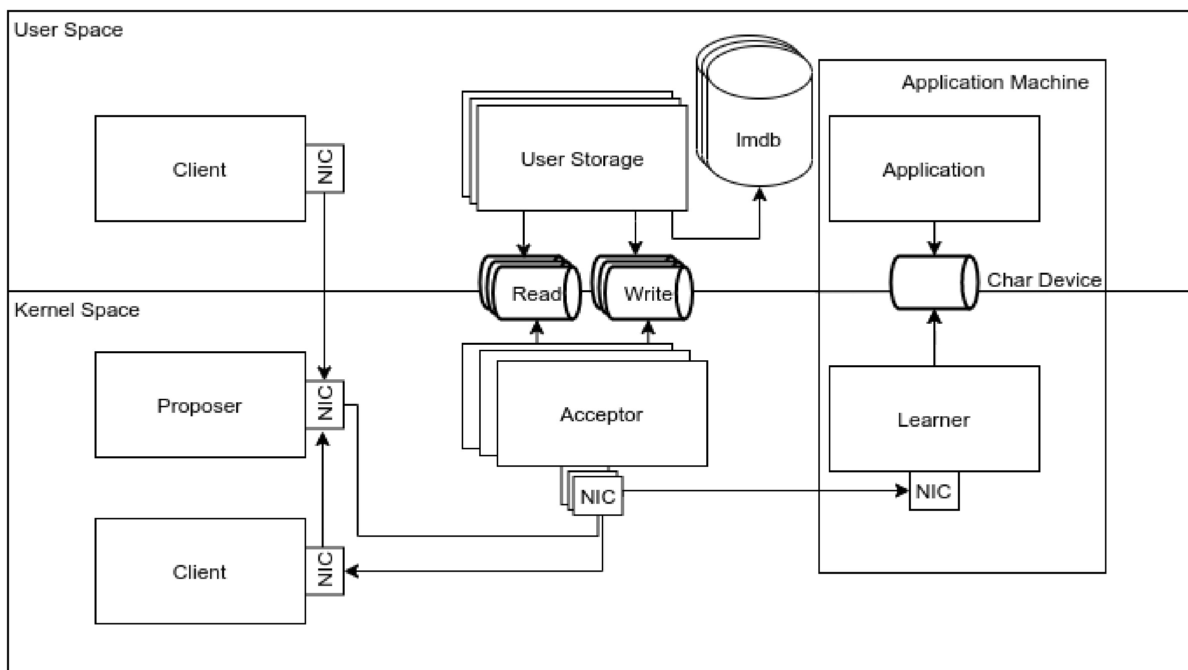


Figura 11 – Arquitetura completa do Kernel Paxos Com Persistência.

4 Resultados

Os experimentos consistiram em executar o Kernel Paxos e o *Libpaxos*, ambos utilizando persistência em disco e em memória. Em todos os modos foram utilizados uma variação de clientes desde 1 a 256, forçando sempre o stress das duas implementações de Paxos.

Para montar o ambiente foram utilizadas 6 máquinas no total, cada uma rodando apenas um único papel do protocolo de Paxos com exceção da máquina *client*. Ficando com a seguinte configuração 1 *proposer*, 3 *acceptors* e 2 *learners*, sendo dentre estes *learners* um também *client*.

Em todo processo a máquina *client* envia valores para o *proposer*, que ao final da execução do algoritmo era recebido pela mesma máquina *client* por ser também um *learner*, possibilitando assim validar a latência entre o envio de um valor e aprendizagem do valor decidido, tal como o número de mensagens por segundo, nos fornecendo o *throughput* do processo.

As máquinas utilizadas nos experimentos possuem a seguinte configuração, 1 vCPU, 2GB de memória RAM, 20GB de disco (HD - Hard Drive) e todas utilizavam Ubuntu 16.04 com *Kernel* 4.4.

Em todos os gráficos comparativos de latência e *throughput* a seguir foi utilizado a média da latência, devido ao libpaxos fornecer apenas a média.

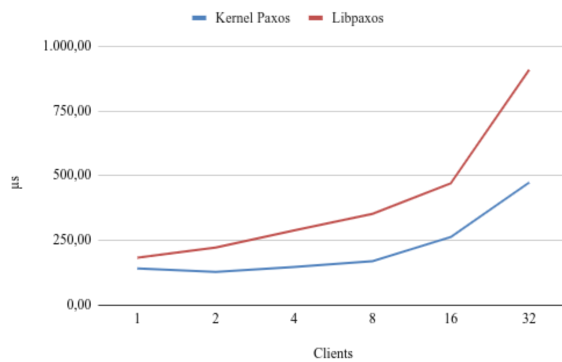
Os gráficos comparativos entre latência e *throughput* (Figuras 12c, 12d, 13c, 13d, 14c e 14d) utilizam a latência no eixo y e o *throughput* no eixo x, por esse motivo é possível ver algumas curvas irregulares, com eles são possíveis identificar a relação entre às duas métricas desconsiderando a quantidade de clientes.

4.1 Análise de Persistência em Memória

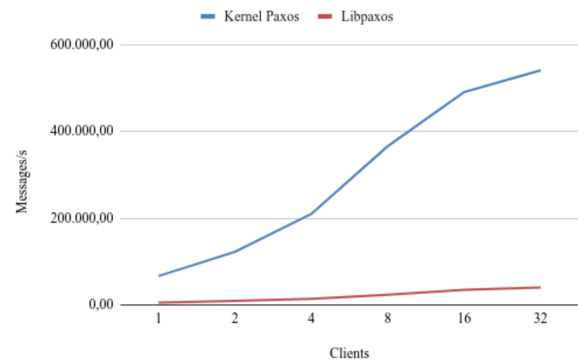
A Figura 12 apresenta um comparativo das duas implementações utilizando persistência em memória. É possível notar o aumento no *throughput* do Kernel Paxos (Figura 12b). Ao chegar em 32 clientes o Kernel Paxos atingiu seu limite e a partir deste ponto vemos a queda nos valores além do aumento na latência (Figura 12a), mas continuando com resultados melhores do que o *libpaxos*. Esta melhora na performance é resultado das otimizações implementadas pelo Kernel Paxos, como a redução do *overhead* de troca de contexto e a utilização de um quadro *ethernet* customizado.

A *throughput* máxima do Kernel Paxos em memória é de 540,8 mil mensagens por

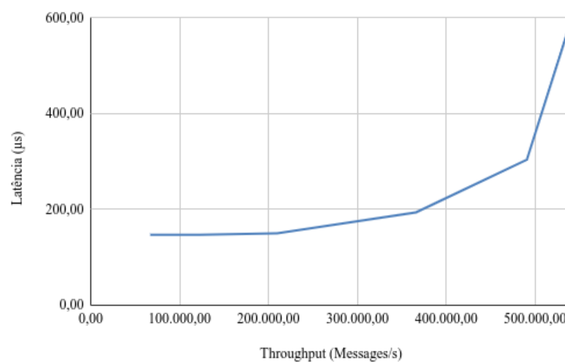
segundo, contra 60,5 mil do *libpaxos*, ou seja, quase 9 vezes maior.



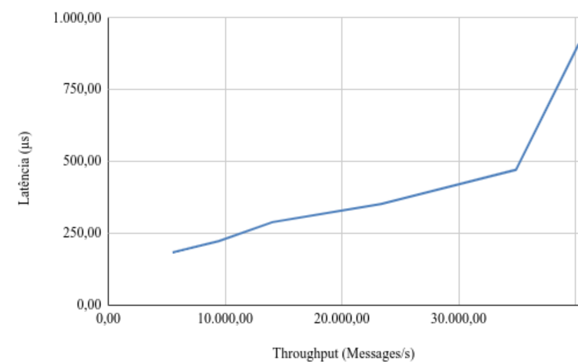
(a) Média de latência da persistência em memória.



(b) Média do *Throughput* da persistência em memória.



(c) Comparativo entre *Throughput* (eixo X) e latência (eixo y) do Kernel Paxos.



(d) Comparativo entre *Throughput* (eixo X) e latência (eixo y) do LibPaxos.

Figura 12 – Experimentos com Persistência em Memória.

4.2 Análise de Persistência em Disco

Utilizando persistência em disco no Kernel Paxos, temos um *buffer* que tem o seu tamanho parametrizado, para estes testes utilizamos tamanho 100 e 1000.

O *lmdb* nos permite configurar a forma que ele realizará a escrita no disco, como assíncrono ou síncrono. De forma geral, no modo síncrono, sempre que é feito uma escrita o *lmdb* persiste diretamente no disco no momento do *commit*. Já o modo assíncrono, ao se fazer uma escrita ela é mantida pelo *lmdb* em memória até que a aplicação solicite explicitamente a persistência ou o espaço reservado em memória se esgote.

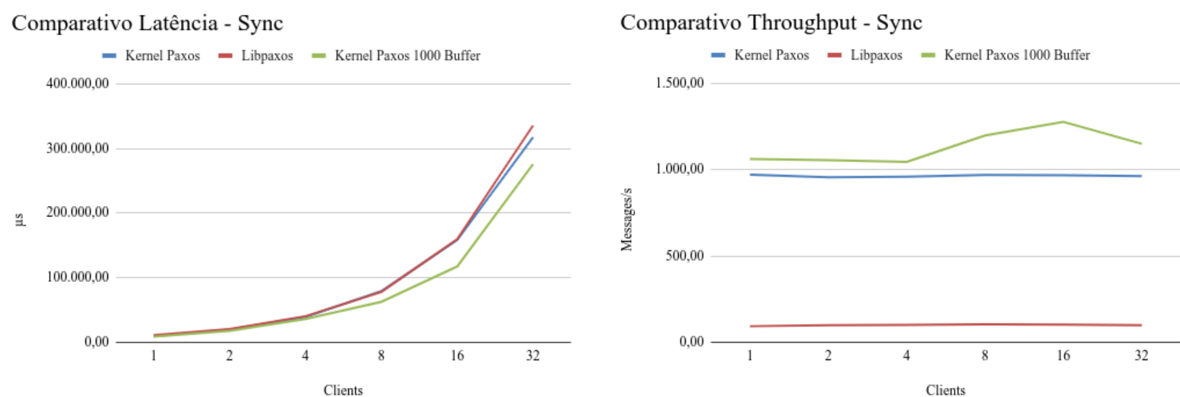
As duas estratégias de persistência obrigam a fazer uma escolha entre consistência e performance. Utilizar o modo assíncrono resultará em uma maior performance por utilizar menos o disco, mas consequentemente em um cenário de falha do *lmdb* os dados podem

ser perdidos por ainda não estarem persistidos. Já utilizando o modo síncrono, ganhamos consistência, mas perdemos performance, pois cada operação é feita utilizando o disco.

4.2.1 Análise de Persistência em Disco Síncrono

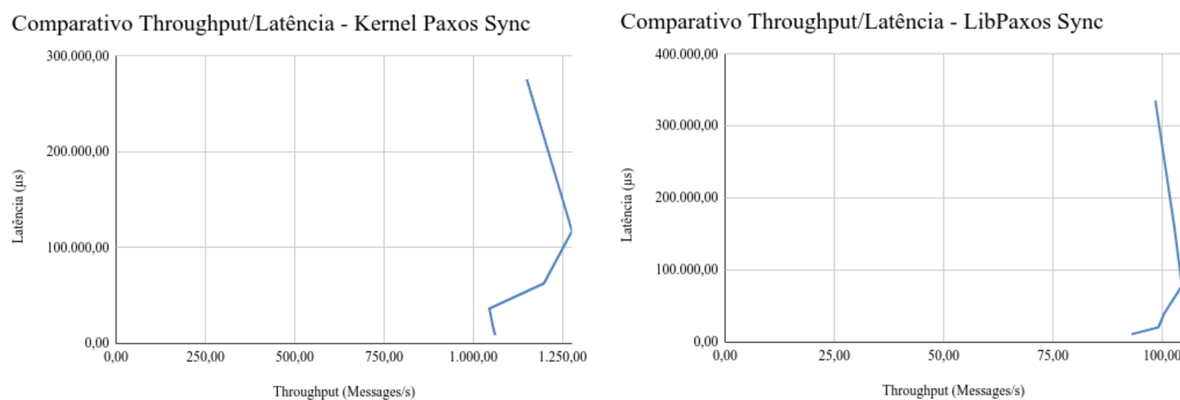
Novamente vemos na Figura 13 um ganho significativo em relação ao *Libpaxos*. No comparativo de *throughput* (Figura 13b) podemos ver uma constância nos valores para diferentes quantidades de clientes. Esse comportamento se deve ao fato de estarmos limitados agora ao tempo de escrita e limites do disco, visto que cada operação aguarda a finalização no disco. Então os processos estão trabalhando com um relativamente baixo uso de CPU quando comparado ao disco. É possível notar também um leve ganho com o aumento do *buffer* dos nossos *Char Devices*.

Com um *throughput* máximo em 1,3 mil o Kernel Paxos atingiu um aumento de 13 vezes o *throughput* do *libpaxos*.



(a) Média de latência da persistência em disco síncrona.

(b) Média do *Throughput* da persistência em disco síncrona.



(c) Comparativo entre *Throughput* (eixo X) e latência (eixo y) do Kernel Paxos.

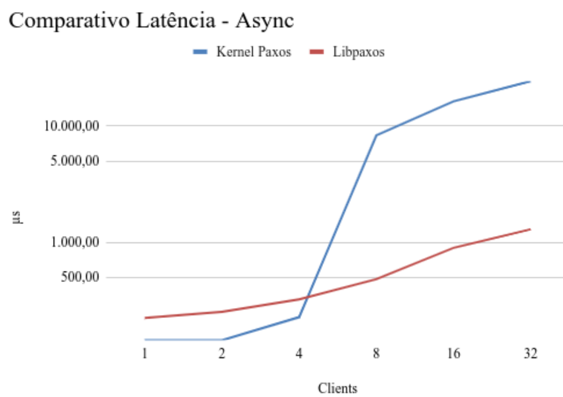
(d) Comparativo entre *Throughput* (eixo X) e latência (eixo y) do LibPaxos.

Figura 13 – Experimentos com Persistência em Disco Síncrono.

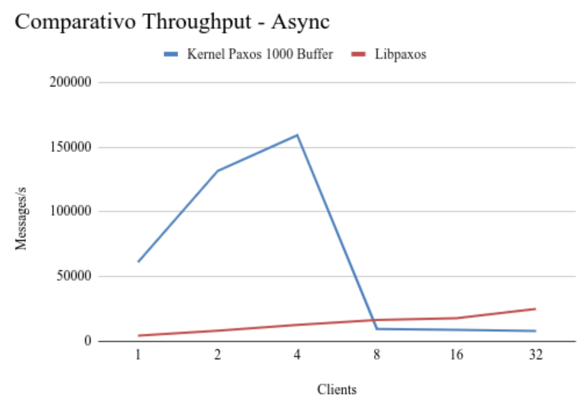
4.2.2 Análise de Persistência em Disco Assíncrono

Pelo motivo de o *buffer* de tamanho 1000 ser mais performático que de tamanho 100, como vimos no experimento síncrono, este último foi retirado deste experimento. Quando trabalhamos com a persistência de forma assíncrona, podemos notar um ganho do Kernel Paxos de 30% em latência (Figura 14a) e 12 vezes em *throughput* (Figura 14b) com 4 clientes, 43% em latência (Figura 14a) e 16 vezes em *throughput* (Figura 14b) para 2 clientes. Conforme visto na Figura 14. O Kernel Paxos neste modo saturou ao atingir 4 clientes, o que fez com que os resultados ao utilizar mais clientes ficassem bem abaixo da *Libpaxos*. Com a saturação o *throughput* (Figura 14b) caiu enquanto a latência subiu (Figura 14a).

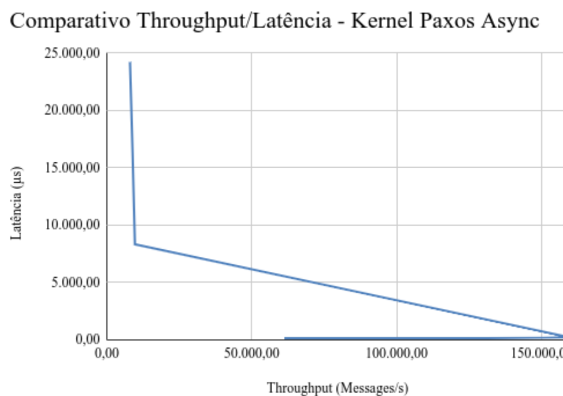
Devido a grande diferença entre as latências no Kernel Paxos foi utilizado uma escala logarítmica para o eixo da latência, nos dois gráficos que ilustram os resultados do Kernel Paxos modo assíncrono.



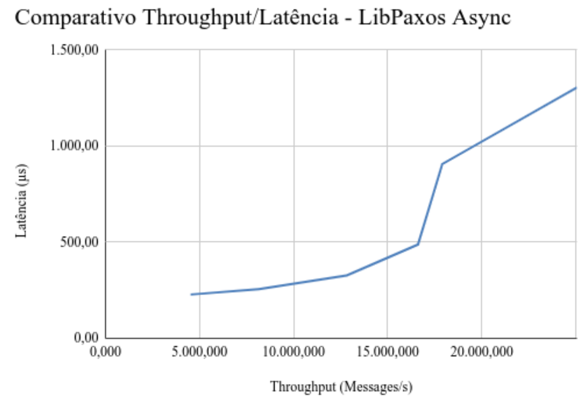
(a) Média de latência da persistência em disco assíncrona.



(b) Média do *Throughput* da persistência em disco assíncrona.



(c) Comparativo entre *Throughput* (eixo X) e latência (eixo y) do Kernel Paxos.



(d) Comparativo entre *Throughput* (eixo X) e latência (eixo y) do LibPaxos.

Figura 14 – Experimentos com Persistência em Disco Assíncrono.

4.3 Resultados - Conclusão

A implementação da persistência no Kernel Paxos mostrou que mesmo inserindo a base de dados no *user space* os ganhos do Kernel Paxos sobre uma implementação de Paxos no *user space*, continuam sendo muito significativos. Ter que escrever ou ler informações no *user space* acaba inserindo um pouco do *overhead* da troca de contexto, mas todo o trabalho dos outros papéis além do *Acceptor* estão livres deste *overhead*, isto somado ao ganho da otimização na comunicação na rede continuam sendo extremamente performáticos.

5 Conclusão

Com este trabalho foi possível, estudar como o Algoritmo de Paxos nos fornece uma tolerância a falhas e consequentemente uma maior disponibilidade, reduzindo a probabilidade de uma parada completa do sistema.

Adicionar o requisito de durabilidade no Kernel Paxos, o que permite uma recuperação em caso de falhas de réplicas no sistema, mostrou-se desafiador e consideravelmente complexo, todo o processo dentro do *Kernel* possui uma grande responsabilidade, e por isto muito cuidado com o seu desenvolvimento, um simples erro pode causar um *Kernel Panic* que interrompe a execução do Sistema Operacional. Para a implementação da persistência, foi necessário um estudo do desenvolvimento e arquitetura do *Kernel Linux*, um estudo aprofundado do Protocolo de Paxos, além de melhores práticas para tal implementação.

Durante todo o desenvolvimento foram encontrados diversos problemas pelo caminho, desde vazamentos de memória, onde um único trecho de código realizando alocação de memória que não era liberada acabava se tornando um grande problema por se tratar de um alto volume de mensagens trocadas, até mesmo uma saturação dos *buffers* dos *char devices*, que exigiu a implementação do processamento em lote. No caso do vazamento de memória, por se tratar do *Kernel*, mesmo realizando o reinício do processo a memória não era liberada, forçando um reinício de todo o Sistema Operacional.

Ao final desta implementação é possível visualizar o ganho que o Kernel Paxos nos dá, tornando o protocolo de Paxos implementado no *Kernel* altamente performático, além de segurança de retornar ao estado em que parou quando é necessário um reinício.

Para trabalhos futuros, há a necessidade de melhorar a saturação que ocorre ao utilizar a persistência no modo assíncrono, a implementação do processamento em lote causou uma saturação com 4 clientes, onde sem tal implementação, ocorria com 2 clientes. Seria importante avaliar também a performance do processo descrito na Seção 3.3.1 isoladamente do *Kernel Paxos*.

Referências

M. Barborak, A. Dahbura, and M. Malek. The consensus problem in fault-tolerant computing. *ACM Computing Surveys*, 25(2):171–220, jun 1993. Citado na página 9.

H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé. NetPaxos. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research - SOSR '15*, pages 1–7, New York, New York, USA, 2015. ACM Press. Citado na página 9.

E. G. Esposito, P. Coelho, and F. Pedone. Kernel Paxos. In *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*, pages 231–240. IEEE, oct 2018. Citado 6 vezes nas páginas 9, 12, 14, 17, 18 e 20.

R. Guerraoui and M. VukoliĆ. Refined quorum systems. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing - PODC '07*, page 119, New York, New York, USA, 2007. ACM Press. Citado na página 9.

G. Kroah-Hartman. Things you should never do in the kernel. *Linux Journal*, 2005. Citado na página 20.

L. Lamport. Paxos Made Simple. *Sigact News - SIGACT*, 32, 2001. Citado 5 vezes nas páginas 9, 10, 13, 15 e 17.

L. Lamport. Generalized Consensus and Paxos. Technical Report MSR-TR-2005-33, 2005. Citado na página 9.

L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, jul 1982. Citado na página 8.

J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports. Just Say {NO} to Paxos Overhead: Replacing Consensus with Network Ordering. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 467–483, Savannah, GA, 2016. {USENIX} Association. Citado na página 9.

I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in Egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP '13*, pages 358–372, New York, New York, USA, 2013. ACM Press. Citado na página 9.

O'Reilly. Chapter 3. char drivers. <https://www.oreilly.com/library/view/linux-device-drivers/0596000081/ch01s03.html>. Accessado: 08-06-2021. Citado na página 21.

Parisa Jalili Marandi, M. Primi, N. Schiper, and F. Pedone. Ring Paxos: A high-throughput atomic broadcast protocol. In *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, pages 527–536. IEEE, jun 2010. Citado na página 9.

S. Peluso, A. Turcu, R. Palmieri, G. Losa, and B. Ravindran. Making Fast Consensus Generally Faster. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 156–167. IEEE, jun 2016. Citado na página 9.

F. B. Schneider. The state machine approach: A tutorial. In *Fault-Tolerant Distributed Computing*, pages 18–41, Berlin/Heidelberg, 2006. Springer-Verlag. Citado na página 11.