UNIVERSIDADE FEDERAL DE SANTA CATARINA
CAMPUS FLORIANÓPOLIS
CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Bernardo Schmidt Farias

Estudo e implementação de estratégias para otimização de E/S em sistemas distribuídos de alta vazão, com foco em técnicas de Kernel-Bypass utilizando DPDK e mTCP

Florianópolis
2023

Bernardo Schmidt Farias

Estudo e implementação de estratégias para otimização de E/S em sistemas distribuídos de alta vazão, com foco em técnicas de Kernel-Bypass utilizando DPDK e mTCP

Trabalho de Conclusão de Curso do Curso de Graduação em Ciência da Computação do Campus Florianópolis da Universidade Federal de Santa Catarina para a obtenção do título de bacharel em Ciência da Computação.
Orientador: Prof. Odorico Mendizabal, Dr.

Florianópolis
2023

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

Farias, Bernardo Schmidt
Estudo e implementação de estratégias para otimização de E/S em sistemas distribuídos de alta vazão, com foco em técnicas de Kernel-Bypass utilizando DPDK e mTCP / Bernardo Schmidt Farias ; orientador, Odorico Machado Mendizabal, 2023.
77 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Ciências da Computação, Florianópolis, 2023.

Inclui referências.

1. Ciências da Computação. 2. Sistemas Distribuídos. 3. Kernel-Bypass. I. Mendizabal, Odorico Machado. II. Universidade Federal de Santa Catarina. Graduação em Ciências da Computação. III. Título.

Bernardo Schmidt Farias

Estudo e implementação de estratégias para otimização de E/S em sistemas distribuídos de alta vazão, com foco em técnicas de Kernel-Bypass utilizando DPDK e mTCP

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de "bacharel em Ciência da Computação" e aprovado em sua forma final pelo Curso de Graduação em Ciência da Computação.

Florianópolis, 07 de Dezembro de 2023.

Lúcia Helena Martins Pacheco, Dra.
Coordenadora do Curso

Banca Examinadora:

Prof. Odorico Mendizabal, Dr.
Orientador

Prof. Marcio Bastos Castro, Dr.
Avaliador
Instituição UFSC

Prof. Wyllian Bezerra da Silva , Dr.
Avaliador
Instituição UFSC

Este trabalho é dedicado aos meus queridos pais e meus queridíssimos amigos e colegas de curso.

AGRADECIMENTOS

A meus pais, meus amigos, meus colegas, meu orientador e a mim. Obrigado.

"The Industrial Revolution and its consequences have been a disaster for the human race. They have greatly increased the life-expectancy of those of us who live in "advanced" countries, but they have destabilized society, have made life unfulfilling, have subjected human beings to indignities, have led to widespread psychological suffering (in the Third World to physical suffering as well) and have inflicted severe damage on the natural world. The continued development of technology will worsen the situation. It will certainly subject human beings to greater indignities and inflict greater damage on the natural world, it will probably lead to greater social disruption and psychological suffering."

(Ted Kaczynski, 1995)

RESUMO

Na última década, os dispositivos físicos de E/S aceleraram sua velocidade de operação enquanto a performance dos processadores se manteve estagnada. Percebe-se uma tendência, principalmente em ambientes de intenso fluxo de dados como os *datacenters*, na qual estes dispositivos não são mais o grande gargalo de performance nas operações de E/S de sistemas distribuídos de alta vazão, mas sim o processamento de rede disponibilizado pelo sistema operacional no qual a aplicação está sendo executada. Tendo isso em vista, técnicas para desviar o processamento de dados do *kernel* do sistema operacional, chamadas de Kernel-Bypass, vêm conquistando interesse de pesquisadores e iniciativas privadas. Este tipo de estratégia almeja remover do *kernel* a responsabilidade de processamento de dados em rede, transferindo esta função para bibliotecas implementadas em espaço de usuário e/ou dispositivos físicos específicos para este fim. O presente trabalho apresenta um estudo sobre os principais gargalos do *kernel* Linux no processamento de E/S para sistemas distribuídos, e como as implementações de estratégias de Kernel-Bypass otimizam estes fluxos. Ao realizar a implementação de protótipos, utilizando em junção duas bibliotecas de Kernel-Bypass, foi possível avaliar o potencial deste tipo de estratégia quando comparado com o fluxo tradicional do *kernel*. Os resultados sugerem que as bibliotecas de Kernel-Bypass utilizadas conseguem extrair maior proveito do *hardware* disponibilizado em certos cenários quando comparadas com o *kernel* Linux. Entretanto, não são recomendadas para implementação de sistemas que não antecipam um fluxo intenso de troca de dados.

Palavras-chave: Sistemas distribuídos de alta vazão, E/S. Kernel-Bypass. DPDK. mTCP.

ABSTRACT

Over the past decade, I/O devices have sped up while CPU performance has stagnated. There's a trend, especially in high data throughput environments like datacenters, that these devices are no longer the main performance bottleneck of I/O operations in high throughput distributed systems, this position is now held by the network stack processing available in Operational Systems kernel. Bearing that in mind, the discussions about techniques that bypass the kernel in the datapath processing of applications, called Kernel-Bypass techniques, are gaining more attention from researchers and companies. This kind of strategy aims to remove from the kernel the responsibility of network processing, offloading this task to libraries implemented in user space and/or hardware specific for this kind of processing. This work offers a study about the main bottlenecks in I/O processing for distributed systems made by the Linux kernel, and how the Kernel-Bypass approach optimizes these flows. In order to validate these strategies, test applications were implemented using two Kernel-Bypass libraries and its performance was compared with the linux Kernel. The results suggests that the Kernel-Bypass libraries used are able to better utilize the given hardware in some scenarios than the Linux kernel network stack. However, they're not recommended for developing systems that do not anticipate intense data flow.

Keywords: High throughput distributed systems. I/O. Kernel-Bypass. DPDK. mTCP.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

AMD	Advanced Micro Devices
API	Application Programming Interface
ARM	Advanced RISC Machines
CPU	Central Processing Unit
DMA	Direct Memory Access
DMAC	DMA Controller
DPDK	Data Plane Development Kit
E/S	Entrada e Saída
GCC	GNU Compiler Collection
GVE	Google Cloud VMware Engine
IP	Internet Protocol
LAN	Local Area Network
mTCP	Multicore TCP
MTU	Maximum Transmission Unit
NUMA	Non-Uniform Memory Access
OSI	Open Systems Interconnection
PCI-e	Peripheral Component Interconnect
PMD	Poll Mode Driver
PPC	Power PC
RDMA	Remote Direct Access Memory
ROCE	RDMA Over Converged Ethernet
TCP	Transmission Control Protocol
TLB	Translation Lookaside Buffer
UDP	User Datagram Protocol

SUMÁRIO

1 INTRODUÇÃO

Em 2023, 5,3 bilhões de pessoas estarão conectadas à Internet (66% da população global), o número de dispositivos conectados a redes Internet Protocol (IP) será mais do que o triplo da população mundial e quase 300 bilhões de aplicativos para celulares serão instalados (CISCO, 2023). Tendo em vista estas estatísticas, é seguro afirmar que nos últimos anos sistemas distribuídos e aplicativos web conquistaram uma permeabilidade significativa na sociedade, tornando-se indispensáveis para certos modelos de negócio e estilos de vida. Esta permeabilidade implica em um interesse de grande parte da população para que esse tipo de sistema seja cada vez mais performático, resiliente, robusto e confiável (LOWE, 2003).

Com o passar do tempo e a popularização dos sistemas web, notou-se um movimento de migração destes para ambientes de computação em nuvem. Ambientes de computação em nuvem provêm gerenciamento mais granular e eficiente de recursos do que estratégias *in-house*, nos quais a empresa mantém e gerencia todo o *hardware* necessário para a execução de aplicações (ALAM, 2020). De acordo com (LOUKIDES, 2023), 48% das empresas possuem planos de migrar 50% ou mais de suas aplicações para ambientes em nuvem. Este tipo de ambiente possui um paradigma naturalmente distribuído, oferecendo ao usuário conexão transparente entre diferentes serviços por meio de comunicação em rede (ASSUNCAO *et al.*, 2014). Além disso, armazenando em disco e trafegando pela rede uma quantidade exorbitante de dados, sendo assim, a performance de rede em latência e largura de banda são cruciais nestes ambientes (FIRESTONE *et al.*, 2018).

1.1 MOTIVAÇÃO

Em 2017, o pesquisador brasileiro Luiz Barroso lançou um desafio para a comunidade científica de computação sobre otimização de operações a nível de microssegundos (BARROSO *et al.*, 2017). Em seu artigo, é alegado que enquanto otimizações para operações em nanosegundos, como por exemplo, *branch predicition*, execução fora de ordem, e para operações de milissegundos, como troca de contexto a nível de software enquanto uma operação lenta é executada, ganham bastante atenção da comunidade científica, eventos com latência na faixa dos microssegundos têm sido deixados de lado.

Sistemas operacionais modernos necessitam de *kernels* de uso abrangente, capazes de serem executados entre diversos dispositivos; essa decisão de projeto generalista tende a deixar princípios como performance em segundo plano (ZHANG; RAYBUCK *et al.*, 2021). Dispositivos de armazenamento e de rede de última geração já atuam na faixa de latência de um dígito de microssegundo, enquanto isso, sistemas operacionais existentes não acompanham essa velocidade. Como consequência, *kernels* tradicionais acarretam um gargalo impagável para aplicações distribuídas de alta vazão que almejam extrair total aproveitamento do *hardware* disponibilizado, como por exemplo os *datacenters* de prove-

dores de computação em nuvem (ZHANG; RAYBUCK *et al.*, 2021). Para exemplificar, as placas de rede que são utilizadas em *datacenters* atuais possuem uma capacidade de transferência de 100 Gitabits por segundo sobre *ethernet*, entretanto estudos recentes mostram que o processamento de rede nativo de sistemas operacionais existentes não consegue acompanhar a velocidade destas placas (BELKHIRI *et al.*, 2023).

As abstrações de E/S existentes (e.g., `socket` Application Programming Interface (API)) possuem um *design* da época em que dispositivos de E/S eram lentos e a aplicação desperdiçaria muito tempo caso esperasse as operações de E/S encerrarem. Atualmente, em contrapartida, dispositivos de E/S de *datacenters* entregam requisições de Entrada e Saída (E/S) mais rápido do que as aplicações conseguem processar. Como por exemplo, o banco de dados em memória Redis (REDIS, 2023), que gasta em média 2µs para processar uma requisição de leitura; para acompanhar a evolução na velocidade das placas de rede modernas, nenhuma latência no processamento poderia ser adicionada pelo sistema operacional (ZHANG; LIU *et al.*, 2019). Presencia-se uma inversão no conhecimento comum, os dispositivos físico não são mais o grande gargalo no processamento de E/S, mas sim operações de processamento de rede existentes em *kernels* tradicionais.

Tendo em vista este cenário, estratégias e dispositivos que buscam desviar o fluxo de processamento de dados do *kernel* têm se tornado cada vez mais atraentes como matéria de estudo. Estes dispositivos e sistemas, popularmente conhecidos como Kernel-Bypass, ignoram o sistema operacional no fluxo de dados por meio da rede, passando responsabilidades relevantes antes cuidadas pela *stack* de rede do sistema operacional, como por exemplo recebimento e envio de pacotes Transmission Control Protocol (TCP), para o dispositivo/sistema de Kernel-Bypass. Ao tomar como abordagem sistemas de Kernel-Bypass, permite-se que as aplicações realizem operações de E/S diretamente do espaço de usuário, ignorando operações notoriamente conhecidas por degradar a latência, como trocas de contexto e cópia dos dados entre espaço de usuário e *kernel*.

Neste trabalho foi feito um estudo sobre os principais gargalos de processamento de E/S existentes no *kernel* Linux, ao mesmo tempo em que se explorava possíveis estratégias de otimização para estas operações em sistemas distribuídos de alta vazão. Ao focar os estudos nas estratégias de Kernel-Bypass, foram escolhidas as bibliotecas Multicore TCP (mTCP) e Data Plane Development Kit (DPDK) para realizar a implementação de um protótipo de sistema distribuído com o propósito de validar o potencial destas bibliotecas no aumento de performance. Ao final, os resultados indicam que o protótipo implementado utilizando as bibliotecas de Kernel-Bypass possuem uma vazão superior no processamento de pequenos pacotes em cenários de alta concorrência quando comparado com o protótipo que utiliza o processamento de rede tradicional disponibilizado pelo *kernel* Linux. Além disso, a aplicação Kernel-Bypass foi capaz de saturar a largura de banda da placa de rede utilizada, alcançando taxa de transferência de 10Gb/s, enquanto a aplicação tradicional não foi capaz do mesmo feito. Por fim, salientou-se que a aplicação desenvolvida utilizando

as bibliotecas de Kernel-Bypass não se comporta bem em cenários de baixa concorrência, oferecendo uma latência superior a aplicação tradicional. Por isso, acredita-se que o uso deste tipo de estratégia deva ser reservado para aplicações que antecipam intenso fluxo de dados e alto grau de concorrência.

1.2 OBJETIVOS

1.2.1 Objetivo geral

Revisar os gargalos conhecidos no processamento de rede disponibilizado pelo *kernel* Linux, realizando uma comparação de desempenho entre aplicações desenvolvidas utilizando bibliotecas de Kernel-Bypass e desenvolvidas tradicionalmente.

1.2.2 Objetivos específicos

- Utilizar o estratégias de Kernel Bypass para otimizar operações de E/S em Sistemas Distribuídos. Serão investigadas soluções existentes na literatura;
- Avaliar o desempenho, custos e trocas associadas ao uso de estratégias de Kernel-Bypass e outras otimizações em bibliotecas para sistemas distribuídos. Avaliar desempenhos utilizando e não utilizando este tipo de estratégia;
- Realizar a testagem e a validação de um protótipo desenvolvido a partir das bibliotecas mTCP e DPDK em ambiente de *testbed* para computação distribuída.

2 FUNDAMENTAÇÃO TEÓRICA

Ao longo deste capítulo serão apresentadas as bases teóricas para melhor compreensão do presente trabalho.

2.1 DIRECT MEMORY ACCESS

A estratégia de Direct Memory Access (DMA) permite que a transferência de dados entre periféricos e a memória principal possa ser feita sem a intervenção do processador. Sem o uso de DMA, o processador fica totalmente ocupado em operações de leitura/escrita na memória, copiando palavra por palavra da memória para periféricos, e vice-versa (HARVEY, 2023), quando lida-se com sistemas com grande volume de dados, essa abordagem não é suficiente e precisa-se de uma estratégia mais eficiente.

Como parte do processo de transferência de dados utilizando DMA, utiliza-se o DMA Controller (DMAC), que orquestra o processo de transferência de dados da memória. Normalmente, o DMAC recebe uma solicitação de um periférico, como por exemplo uma placa de vídeo, para ter acesso direto à memória principal, o DMAC então solicita à Central Processing Unit (CPU) acesso ao barramento a fim de começar a transferência de dados (OSBORNE, 1980). Após o processo, a CPU é sinalizada pelo DMAC para verificar se o processo foi concluído corretamente. É importante ressaltar que mesmo sem utilizar o processador, durante a transferência de dados o barramento Peripheral Component Interconnect (PCI-e) fica em utilização pelo *hardware* referente a DMA, não podendo ser utilizado pela CPU dependendo do tipo de modo de transferência que está sendo feito. A Figura 1 demonstra de forma esquemática o funcionamento do DMA. Existem 3 principais modos de funcionamento para *chips* DMA. Segue abaixo uma descrição simplificada de cada um deles.

2.1.1 Modo Explosão

Neste modo, o periférico em questão ganha acesso total ao barramento para terminar a transferência de dados o quanto antes. Após o término do processo, os barramentos são liberados novamente para a utilização da CPU. Sua grande vantagem é a velocidade na transferência, já que ocorre toda de uma vez, entretanto, bloqueia o funcionamento da CPU até o fim da transferência;

2.1.2 Roubo de Ciclo

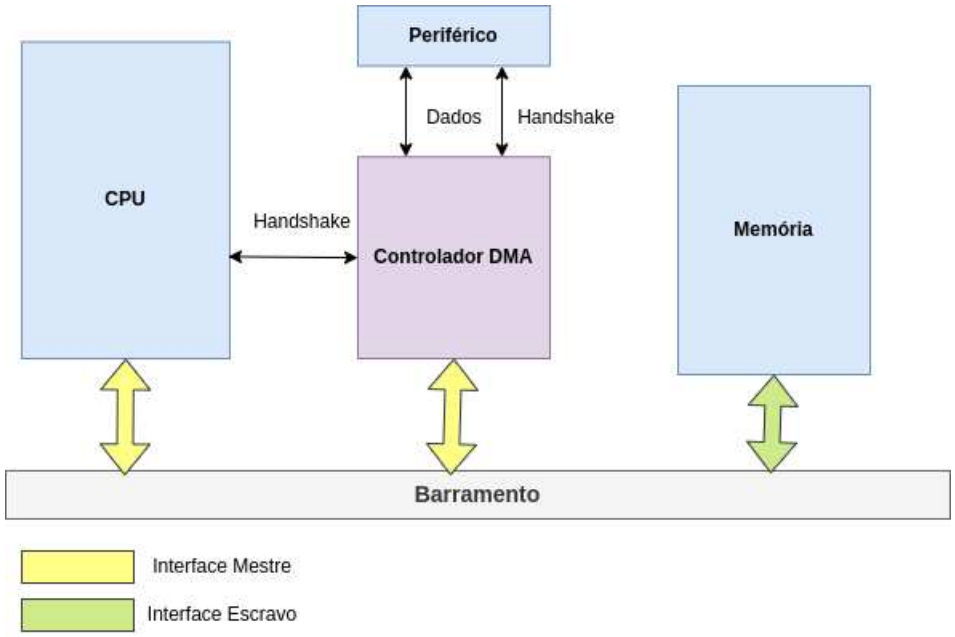
No modo de roubo de ciclo, a transferência é realizada de forma mais lenta. O DMAC solicita a CPU acesso aos barramentos ao final de cada ciclo da CPU, transferindo apenas um byte antes de devolver o controle do barramento a CPU. A cada novo ciclo,

novas solicitações são feitas e novos bytes são transferidos, até que a transferência esteja completa;

2.1.3 Modo Transparente

Extremamente parecido com o Modo Explosão, este modo também dá acesso ao barramento ao periférico para realizar a transferência de uma vez. Entretanto, neste modo, a CPU continua executando operações que não exijam acesso ao barramento. Ao final da transferência, o barramento retorna ao controle da CPU. A Figura 1 oferece

Figura 1 – Demonstração do funcionamento básico de um sistema com DMA



Fonte: Elaborado pelo autor (2023)

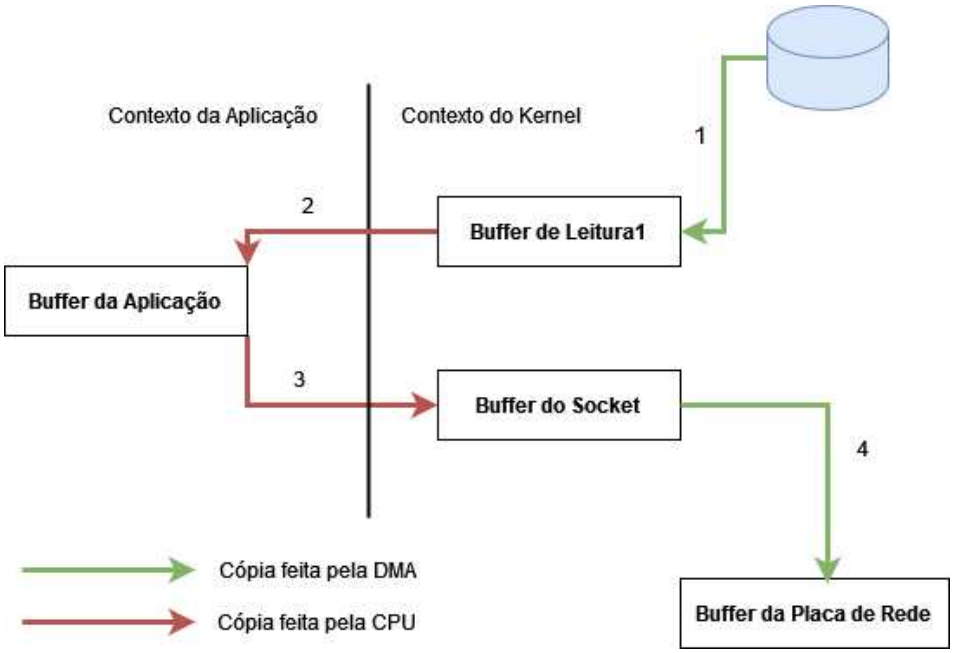
uma visualização simples do funcionamento da estratégia DMA em junção com outros componentes relevantes a um computador, funcionando no modelo Mestre/Escravo.

2.2 ZERO-COPY

A estratégia de Zero-Copy permite a transferência de dados de uma parte da memória para a outra diretamente, dispensando cópias de dados e trocas de contexto desnecessárias. Imagine o funcionamento de um servidor Web moderno; ao receber uma requisição, a aplicação a nível de usuário faz uma leitura no disco para buscar as informações e uma escrita destas informações no *socket* que realizou a requisição. Destrinchando melhor esse fluxo, a aplicação faz uma chamada de sistema ao *kernel*, que então lê as informações do disco e envia de volta ao nível de aplicação, a aplicação então faz outra chamada de sistema, enviando toda a informação lida do disco de volta a nível de *kernel*, escrevendo-a no *socket* que realizou a requisição. Resumidamente, a aplicação neste caso serve como

um intermediário ineficiente, criando apenas etapas desnecessárias na transferência de informação entre o disco e o *socket*. A cada troca de contexto realizada pelas chamadas de sistema, cópias das informações são feitas em *buffers* a nível de *kernel*, essas cópias também consomem ciclos da CPU que poderiam ser poupados. Ao utilizar a abordagem Zero-Copy, a transferência de dados entre o disco e *socket* é feita diretamente a nível de *kernel*, sem precisar passar todas essas informações para o espaço de usuário e depois de volta para o espaço de *kernel*, sendo essa transferência feita diretamente pelo dispositivo DMA e sem a necessidade do uso do processador. Resumidamente, tirando o intermediário da equação e realizando as operações de leitura e escrita diretamente a nível de *kernel*. No *kernel* Linux a chamada de sistema `sendfile` (PAGE, 2023) é disponibilizada para realizar essa operação. Na Figura 2 é demonstrado um fluxo normal de leitura e escrita sem o uso da estratégia de Zero-Copy. Neste fluxo, as informações são copiadas do *kernel* para a aplicação e da aplicação para o *kernel*, sendo realizadas 4 trocas de contexto no total. Já na Figura 3, temos um fluxo utilizando a estratégia Zero-Copy por meio da

Figura 2 – Exemplo do funcionamento sem a abordagem Zero-Copy.



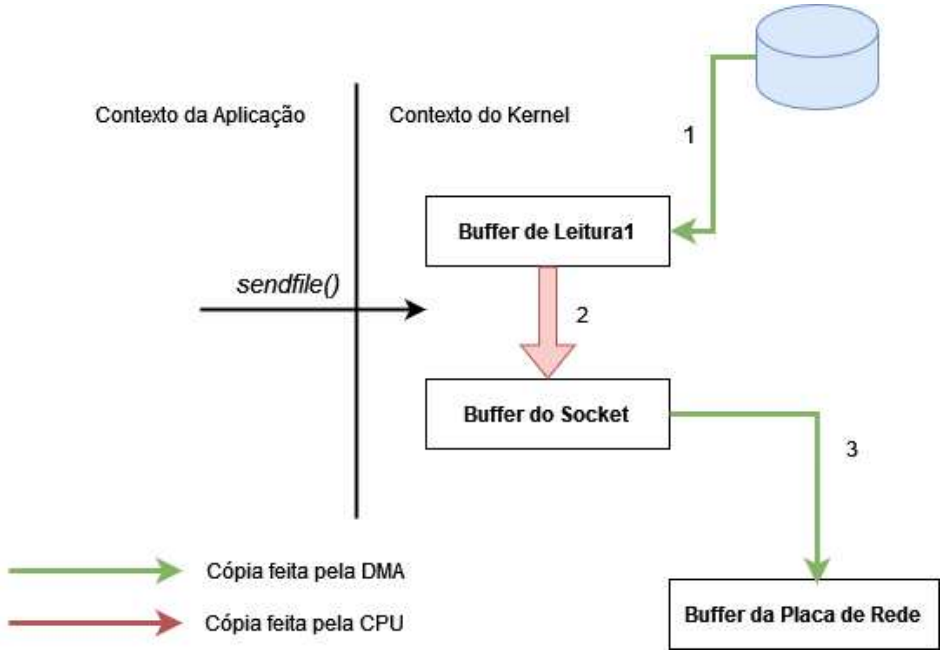
Fonte: Elaborado pelo autor (2023)

chamada de sistema `sendfile` disponibilizada pelo sistema operacional Linux. Neste caso, a cópia está sendo feita direto a nível de *kernel*, realizando apenas 2 trocas de contexto: uma na chamada de sistema `sendfile` e outra no retorno desta chamada.

2.3 KERNEL-BYPASS

Enquanto fabricantes de placas de rede já produzem placas que alcançam até 100 Gigabits de transferência de dados, *kernels* existentes não conseguem suprir a demanda

Figura 3 – Exemplo do funcionamento da abordagem Zero-Copy.



Fonte: Elaborado pelo autor (2023)

de alta velocidade para processamento de pacotes que estas placas exigem, desperdiçando grande parte do potencial do *hardware* (BELKHIRI *et al.*, 2023). Alguns dos principais gargalos conhecidos no processamento de rede são: Chamadas de sistemas, cópias desnecessárias de dados, processamento de pacotes feito por interrupções e alocação dinâmica de memória (CHEN; SUN, 2018). Além disso, de acordo com (PESTEREV *et al.*, 2012), em sistemas *multicores* é comum que múltiplos núcleos do processador interajam no processamento de um pacote e do gerenciamento de uma conexão TCP, causando ineficiências como: invalidações de *cache*, dados sendo copiados entre as *caches* dos núcleos e problemas de performance relacionados a alocação dinâmica de memória, todos estes fatores influenciam negativamente na velocidade de processamento de pacotes. Ainda em máquinas *multicores*, existem estruturas de dados compartilhadas que precisam ser acessadas por diferentes núcleos de processamento, estes acessos devem ser controlados por travas de concorrência e semáforos, que também acabam acarretando em tempo de processamento desperdiçado Tendo em vista a ineficiência dos atuais *kernels* para o processamento de alta vazão de pacotes, pesquisadores tem estudando duas principais abordagens para acelerar este fluxo: aceleração da *stack* de rede já existente dentro do sistema operacional (PESTEREV *et al.*, 2012) e estratégias para desviar o fluxo de processamento de dados do *kernel* para bibliotecas a nível de usuário (CHEN; SUN, 2018; JEONG *et al.*, 2014; INTEL, 2023; RIZZO, 2012), que será o foco deste estudo. As estratégias do segundo tipo são o que chamamos de Kernel-Bypass, ou seja, um conjunto de técnicas que visa desviar o fluxo de processamento de dados do sistema operacional devido a limitações de performance deste para sistemas de alta vazão. Estratégias de Kernel-Bypass mudam

drasticamente arquiteturas tradicionais de Sistemas Operacionais: elas eliminam a atuação do *kernel* no *datapath* de E/S a fim de eliminar sobrecustos operacionais conhecidos no fluxo de processamento de dados pelo *kernel* (ZHANG; RAYBUCK *et al.*, 2021). De acordo com (CHEN; SUN, 2018), as técnicas de Kernel-Bypass se dividem em três classes: bibliotecas para processamento de E/S de pacotes em espaço de usuário, *stacks* TCP/IP desenvolvidas em espaço de usuário e *stacks* de rede híbridas. Entretanto, ainda pode-se adicionar mais uma classe a esta classificação; estratégias que combinam componentes de *hardware* e *software* para obter resultados de performance.

2.3.1 Bibliotecas para processamento de E/S de pacotes a nível de usuário

Estas bibliotecas têm como propósito otimizar o processo de levar um pacote da placa de rede até a aplicação. Por serem feitas a nível de usuário, não possuem muito dos gargalos já comentados anteriormente. Normalmente são empregados mapeamentos de memória da Aplicação para a placa de rede, ou então algum tipo de *driver* específico é disponibilizado para a comunicação com a placa de rede. Por exemplo, a biblioteca DPDK (INTEL, 2023) funciona criando um mapeamento de memória da Aplicação diretamente para a placa de rede e disponibilizando os Poll Mode Driver (PMD), que são *drivers* utilizados para se comunicar com a placa de rede.Assim, removendo o *kernel* por completo da fluxo de processamento, conforme exemplificado pela Figura 6. Já a biblioteca netmap (RIZZO, 2012) segue uma estratégia parecida com DPDK, entretanto utiliza ainda os *drivers* disponibilizados pelo *kernel* para comunicação com a placa de rede. Importante ressaltar que estas bibliotecas não disponibilizam processamento de protocolos de mais alto nível (e.g., TCP/IP), e sim apenas até a camada L2 do modelo Open Systems Interconnection (OSI). Os processamentos de mais alto nível são deixados a encargo da Aplicação.

2.3.2 *Stacks* TCP/IP a nível de usuário

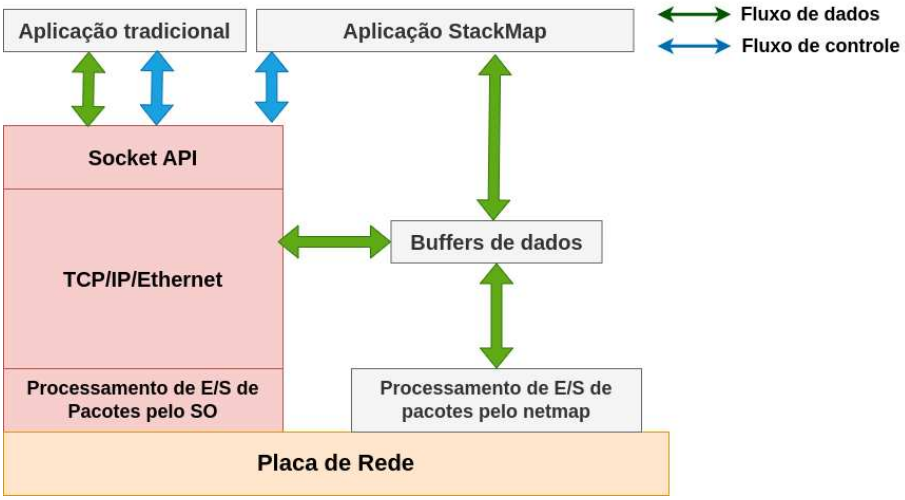
Implementações como mTCP (JEONG *et al.*, 2014) visam reescrever a nível de usuário a *stack* de processamento TCP/IP, a fim de se livrar de sobrecustos operacionais das implementações clássicas disponibilizada pelo *kernel*. Normalmente estas soluções são baseadas em bibliotecas de processamentos de E/S de pacotes em espaço de usuário (discutidas na Seção 2.3.1) para o processamento de mais baixo nível e implementam a parte mais complexa do processamento TCP/IP. No caso da biblioteca mTCP (JEONG *et al.*, 2014), cada núcleo do processador executa uma *thread* mTCP, gerenciando separadamente estruturas de dados como filas de aceitação de conexão e lista de *sockets*, assim, permitindo mais localidade de conexão (CHEN; SUN, 2018). Essa abordagem também permite o uso de estruturas de dados *Lock-free* (LANGDALE, 2023), que eliminam a espera de acesso em estruturas de dados compartilhadas. De acordo com (CHEN; SUN, 2018), a simples

utilização da biblioteca mTCP pode aumentar a performance de aplicações existentes em até 3x.

2.3.3 *Stacks* de rede híbridas

Um dos problemas das estratégias listadas na Seção 2.3.2 é que, mesmo que reescrever do zero uma *stack* de rede a nível de usuário proporcione um aumento considerável em performance, normalmente as *stacks* TCP/IP já existentes nos *kernels* atuais são mais atualizadas e menos limitadas do que estas novas implementações (YASUKATA *et al.*, 2016). Por exemplo, a biblioteca mTCP dá suporte apenas ao protocolo TCP e é simples demais para ser colocada em um ambiente de produção realista (CHEN; SUN, 2018). Assim, surgiu uma categoria híbrida de implementações Kernel-Bypass que visa utilizar toda a robustez da *stack* de rede disponibilizada pelo *kernel*, ao mesmo tempo que utiliza das bibliotecas de processamento de E/S de pacotes em espaço de usuário (descritas na Seção 2.3.1) para otimizar a performance de processamento de camadas mais baixas. A biblioteca

Figura 4 – Representação do funcionamento da biblioteca híbrida StackMap



Fonte: Elaborado pelo autor (2023)

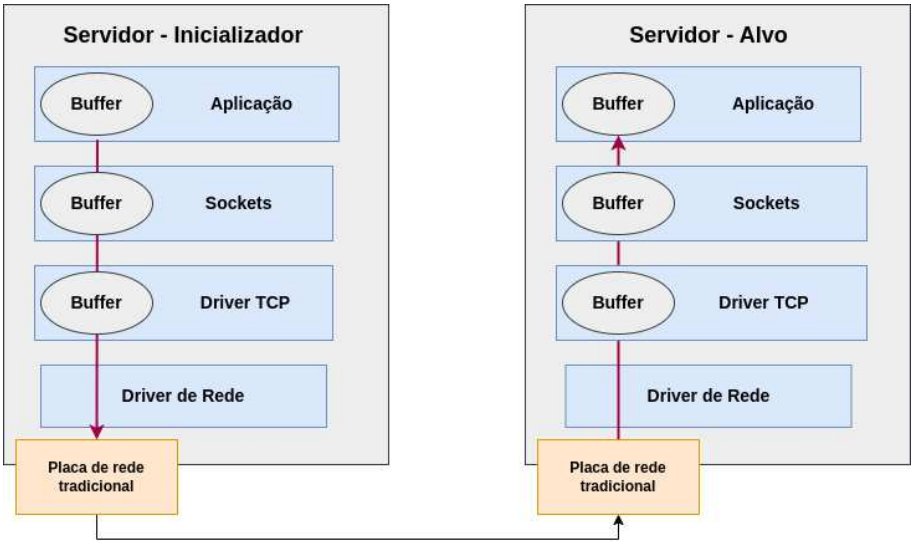
StackMap (YASUKATA *et al.*, 2016) implementa essa estratégia, utilizando da biblioteca netmap (RIZZO, 2012) para o processamento de E/S de pacotes. A Figura 4 exemplifica a estrutura de funcionamento desta biblioteca. Diferente da biblioteca mTCP, ao final do processamento E/S o pacote é enviado para o processamento TCP/IP disponibilizado pela *kernel*. Dessa forma, StackMap consegue aproveitar da melhoria de performance oferecida pela biblioteca netmap, ao mesmo tempo que pode utilizar dos desenvolvimentos mais modernos dos protocolos TCP/IP implementadas no *kernel*.

2.3.4 Combinações de *Hardware* e *Software*

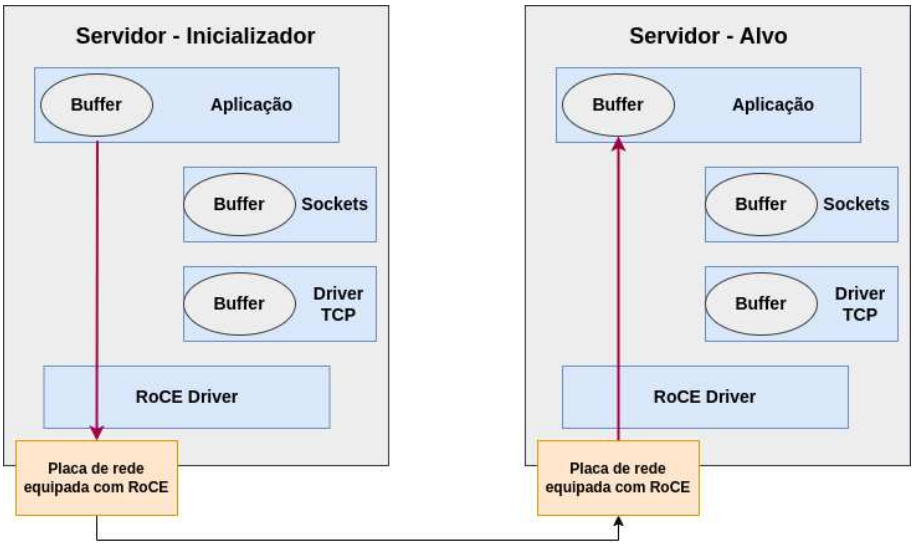
Por último, existe a classe de soluções que utilizam de uma combinação não só de bibliotecas a nível de usuário, conforme as soluções anteriores, mas também de dispositivos de *hardware* para realizar otimizações de Kernel-Bypass. Como principal representante desta classe temos os dispositivos de Remote Direct Access Memory (RDMA). Esta tecnologia permite que dois processos em nodos diferentes na rede possam se comunicar diretamente a nível de memória, sem interferência do processador. Diferente das outras estratégias, esta técnica depende de *hardware* específico para este propósito. As placas de rede RDMA seguem a interface `Verbs` (IETF, 2003) para realizar a comunicação entre si. Estas placas contam com processamento TCP/IP próprio, não dependendo de implementações destes protocolos pelo *kernel* ou a nível de aplicação.

Alguns exemplos de fabricantes são Infiniband e Mellanox, sendo os principais protocolos utilizados para a comunicação entre placas de rede RDMA: RDMA Over Converged Ethernet (ROCE) v2 e iWarp. Na Figura 5 é apresentada uma demonstração de alto nível de como esse processo funciona comparado com o fluxo original. Enquanto na Figura 5a os dados tem que ser trafegados por todo o processamento do *kernel*, na solução RoCE, apresentada na Figura 5b, a comunicação entre aplicações é feita diretamente por meio das placas de rede RDMA, sem intervenção do *kernel*.

Figura 5 – Demonstração do fluxo de dados de um sistema tradicional e um sistema com RDMA, utilizando o protocolo ROCE.



(a) Fluxo de dados tradicional



(b) Fluxo de dados utilizando o protocolo RoCE

Fonte: Elaborado pelo autor (2023)

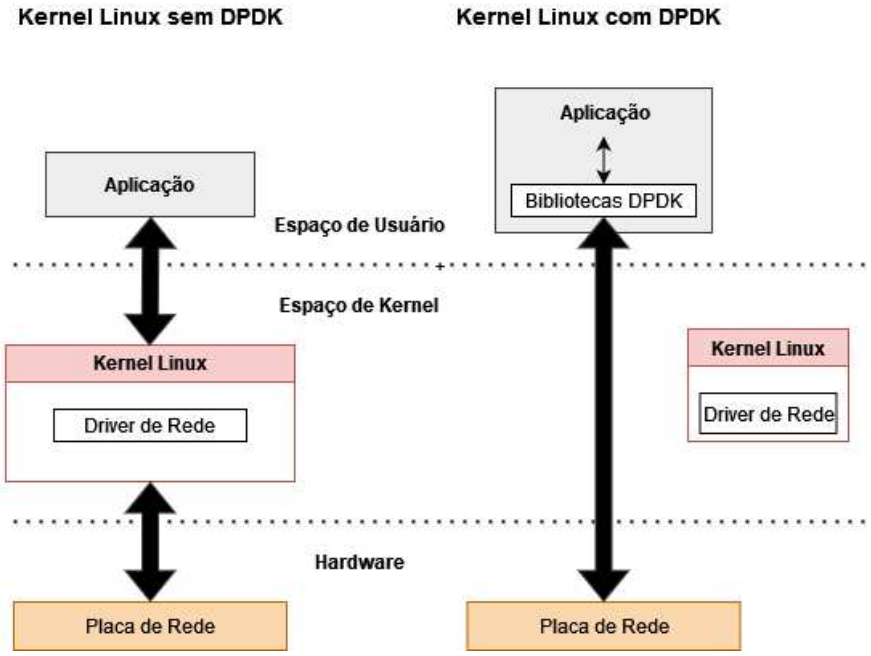
2.4 DATA PLANE DEVELOPMENT KIT (DPDK)

DPDK é um *framework* desenvolvido baseado na estratégia de Kernel-Bypass para acelerar o processamento de E/S de pacotes, a biblioteca implementa este fluxo totalmente a nível de usuário (BELKHIRI *et al.*, 2023). DPDK é um projeto de código aberto gerenciado pela Linux Foundation. Funciona criando uma ligação direta entre uma aplicação a nível de usuário e a placa de rede. Dessa forma, pacotes de rede podem ser direcionados da placa de rede diretamente ao processo a nível de usuário, sem passar por todo o processamento normalmente feito pelo *kernel*. Aplicações construídas a partir da biblioteca DPDK possuem autonomia para escolher como processarão estes pacotes, normalmente optando por implementar protocolos de mais alto nível já conhecidos, como TCP ou User Datagram Protocol (UDP).

Existem alguns passos de configuração necessários para o correto funcionamento do *framework* DPDK. Primeiramente, por trabalhar com o modelo de *polling* por meio dos *drivers* PMD, ao invés de esperar por uma interrupção por parte da placa de rede ao receber pacotes, a CPU constantemente requere a placa de rede novos pacotes para serem processados. O modelo PMD consome mais do potencial de uma CPU devido ao modelo *polling*, entretanto. consegue lidar bem com sistemas de alta vazão de pacotes.

Outro pré-requisito é a configuração de um tamanho maior de páginas na Tabela de Páginas do sistema operacional, chamadas historicamente de *Hugepages*. Ao utilizar alocações de *Hugepages* ao invés do tamanho normal de 4KB, menos páginas são necessárias para o processo em execução, assim, reduzindo o tempo de tradução do endereço virtual para o físico, já que a quantidade de faltas na cache de tradução Translation Lookaside Buffer (TLB) será reduzida. Sem as *Hugepages* a taxa de faltas da TLB tende a aumentar, degradando a performance. Além disso, alocar páginas de tamanho maior faz com que seja menos provável que novas páginas precisem ser alocadas futuramente, poupando assim a aplicação do sobrecusto de alocação de memória.

Figura 6 – Representação de alto nível do funcionamento da biblioteca DPDK em um Sistema Operacional linux



Fonte: Elaborado pelo autor (2023)

2.5 MTCP

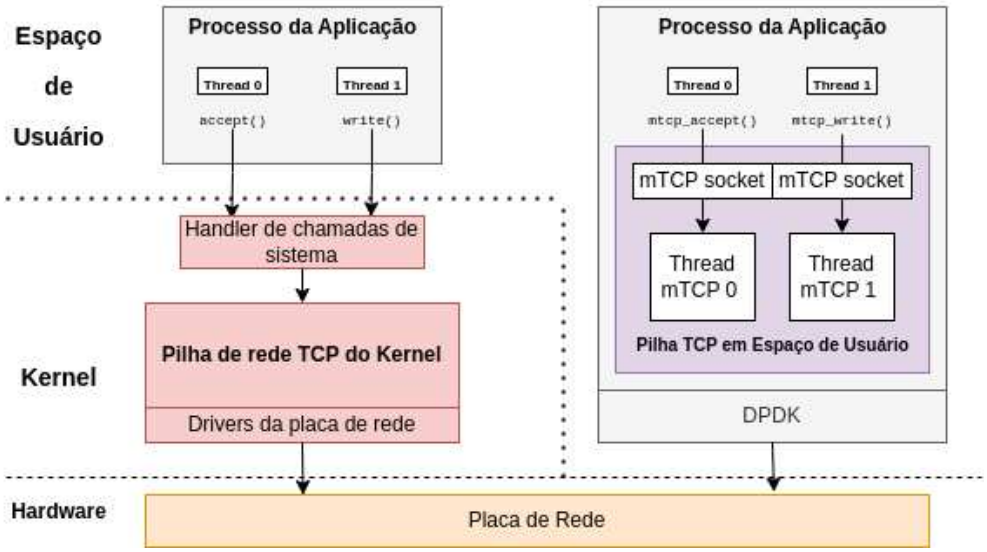
mTCP (JEONG *et al.*, 2014) é uma biblioteca de impletementação da *stack* TCP/IP a nível de usuário, focada em performance de alta vazão e em otimizar pequenas transações TCP em sistemas *multicore* (na obra os autores se referem a "transações" como um par de requisição-resposta). De acordo com (JEONG *et al.*, 2014) 90% de todo o tráfico TCP é menor do que 32KB e mais que metade disto é menor do que 4KB, assim, trabalhos de otimização de transações pequenas teriam possivelmente grande impacto.

Essa biblioteca foca em otimização, empregando técnicas de aceleração conhecidas, como juntar as requisições de processamento e eventos em lotes antes de fazer a troca de contexto entre *threads*, diminuindo assim custos operacionais conhecidos de trocas de contexto frequentes. Além disso, proporciona *sockets* por núcleos de processamento, amortizando possíveis interferências de outros núcleos em uma mesma conexão TCP/IP. Outro grande foco da biblioteca é a fácil portabilidade de sistemas já existentes. mTCP possui uma interface parecida com os *sockets* de Berkley (STEVENS; FENNER; RUDOFF, 2003), API já utilizada pelo *kernel* Linux para comunicação de rede, e requer alterações mínimas no *kernel* para funcionar. Por exemplo, a chamada de sistema `accept()` da interface padrão Linux é substituida pela chamada da função `mtcp_accept()`.

Conforme descrito na Seção 2.3.2, este tipo de biblioteca normalmente opera com uma biblioteca de otimização de E/S de pacotes a nível de usuário, a fim de permitir que a aplicação como um todo não faça a utilização do *kernel* para a transferência e proces-

samento de pacotes de rede. A Figura 7 dá uma visão mais detalhada do funcionamento interno da biblioteca.

Figura 7 – Resumo da arquitetura de mTCP



Fonte: Elaborado pelo autor (2023)

Analisando a figura podemos notar que enquanto grande parte do processamento de pacotes TCP ocorre normalmente em espaço de *kernel* em arquiteturas tradicionais, no caso da biblioteca mTCP, esse processamento é movido para espaço de usuário, sendo realizado pelas *threads* mTCP.

3 TRABALHOS RELACIONADOS

O uso de estratégias de Kernel-Bypass para a otimização de performance em processamento de pacotes é um tema relativamente novo na comunidade científica. Algum empenho na melhoria da *stack* de rede do *kernel* Linux já havia sido realizado (PESTEREV *et al.*, 2012), entretanto esses esforços focavam em melhorar os fluxos de processamento já existentes de dentro do *kernel*, seja essa melhoria por otimizações diretamente a nível de código ou a um nível estrutural e/ou de *design* da *stack* de rede. Quando fala-se de Kernel-Bypass, ao invés de otimizar o já existente fluxo do *kernel*, as estratégias focam na criação de tecnologias e técnicas que desviem o fluxo de processamento de dados do *kernel* e passem esse fluxo para novas camadas de *software/hardware*.

Em (CHEN; SUN, 2018) é feita uma análise do atual estado da arte das principais técnicas de Kernel-Bypass existentes, colocando em comparação com a atual *stack* de rede do *kernel* Linux e alterações já feitas nessa *stack* com propósito de otimização. Após fazer uma revisão do oneroso fluxo de processamento de um pacote pela *stack* de rede disponibilizada atualmente pelo *kernel* Linux, os autores classificam os principais *overheads* deste processamento em três categorias:

1. ***Overhead* de chamadas de sistema:** A cada interação entre a Aplicação e o *socket* de rede, uma chamada de sistema de sistema deve ser feita. Essas chamadas ocasionam trocas de contexto, que por si só já são custosas (TSUNA, 2010), mas ainda podem ocasionar poluição da cache e expurgo da TLB, que também acabam degradando a performance;

2. **Cópias de dados:** Ao receber ou ao enviar pacotes, um pacote de rede é copiado em *buffers* tanto a nível de usuário quanto a nível de *kernel*, essas cópias tomam ciclos da CPU e degradam a performance;

3. **Processamento por pacote:** Cada pacote que chega na placa de rede é processado individualmente pelo modelo de interrupção, a placa de rede manda uma interrupção para o processador, que então processa individualmente este pacote. Como consequência de cada uma dessas interrupções, estruturas de dados pesadas como `sk_buff` (COX, 1996) tem que ser alocadas dinamicamente. Em casos nos quais muitos pacotes estão chegando para processamento, esse tipo de alocação pesada e cópia de dados pode facilmente se tornar um gargalo de processamento.

Tendo estabelecido que a *stack* de rede atual é insuficiente para sistemas de alta vazão, são apresentadas as principais estratégias de Kernel-Bypass, sendo classificadas em três principais classes:

1. **E/S de pacotes em Espaço de Usuário:** Essas estratégias focam na otimização do processamento do pacote a partir do recebimento pela placa de rede até a *stack* do

hospedeiro. Importante ressaltar que soluções desta classe não fazem processamento de protocolos de mais alto nível (e.g., TCP/IP), mas apenas focam na transferência eficiente do pacote da placa de rede até o começo do processamento de protocolos. Exemplos de soluções desta classe são DPDK (INTEL, 2023) e netmap (RIZZO, 2012).

2. ***Stack* TCP/IP em Espaço de Usuário:** Esta classe de soluções foca na otimização do processamento TCP/IP, criando bibliotecas em espaço de usuário para o processamento da *stack* TCP/IP. Estas *stacks* por serem criadas em espaço de usuário e tendo em vista otimização costumam ter uma boa performance, entretanto acabam sendo limitadas quando comparadas a *stack* TCP/IP existente no *kernel*. Essas soluções normalmente se baseiam em bibliotecas de entrega de pacotes em espaço de usuário, descritas no item anterior.

3. ***Stacks* de rede híbridas:** Este tipo de estratégia visa aproveitar um pouco de cada classe de solução descrita anteriormente. Normalmente estratégias nesta classe portam a *stack* TCP/IP já existente do *kernel* para uma biblioteca de alta performance de processamento de pacotes em Espaço de Usuário (como por exemplo DPDK). Dessa forma, conseguem aproveitar da implementação mais robusta da *stack* TCP/IP feita pelo *kernel*, ao mesmo tempo que aproveitam da melhoria de performance graças as bibliotecas de processamento de pacotes

Em (ZHANG; RAYBUCK *et al.*, 2021) a discussão sobre dispositivos Kernel-Bypass é elevada a outro nível. Ao invés de propor uma nova biblioteca/dispositivo para realizar Kernel-Bypass, é levantada a discussão sobre a portabilidade destas novas técnicas para uma interface generalizada.

Estratégias diferentes de Kernel-Bypass implementam diferentes funcionalidades de Sistemas Operacionais já existentes. Por exemplo, a biblioteca de processamento de E/S de pacotes DPDK (INTEL, 2023) implementa alocação de memória e até a etapa de processamento na qual o pacote está pronto para ser executado por protocolos de alto nível, ou seja, toda a implementação de uma *stack* TCP/IP a nível de usuário fica a encargo da aplicação que irá utilizar esta biblioteca para otimização de processamento de pacotes. Já a tecnologia RDMA (IETF, 2003) implementa um protocolo de rede com controle de congestionamento e transmissão confiável de pacotes (ZHANG; RAYBUCK *et al.*, 2021), mas não cuida da alocação de memória dinâmica. Ou seja, sistemas que utilizam DPDK implementam uma *stack* de rede completa, entretanto esta mesma *stack* é desnecessária para sistemas utilizando RDMA

Tendo este dilema em mente, no trabalho é proposto o Demikernel, um Sistema Operacional flexível com foco no *datapath* e com uma arquitetura feita para dispositivos de Kernel-Bypass heterogêneos. A principal ideia do trabalho é possibilitar ao desenvolvedor uma interface comum de implementação ao trabalhar com diversos dispositivos com

competências diferentes de Kernel-Bypass, fazendo *offload* do processamento de certas funcionalidades ao dispositivo sempre que possível. Por exemplo, para o caso da biblioteca DPDK (INTEL, 2023) é oferecida uma *stack* TCP/IP a nível de usuário para suprir a falta desta. Já para dispositivos RDMA (ZHANG; RAYBUCK *et al.*, 2021), é feito o gerenciamento de alocação dinâmica de memória, deixando a cargo do desenvolvedor apenas a lógica da aplicação em si, ao invés de se preocupar com detalhes específicos de cada dispositivo.

Por meio de experimentos práticos, observou-se que utilizando o Demikernel, é possível criar e fazer a portabilidade de aplicações com latências de operações de E/S de aproximadamente 50ns por operação, tendo um custo operacional de 17% a 26% na vazão quando comparado diretamente ao uso dos dispositivos Kernel-Bypass em suas respectivas formas nativas.

Em (JEONG *et al.*, 2014) é feita uma análise profunda sobre a atual *stack* de rede do Linux, apontando alguns dos gargalos conhecidos desta pela comunidade científica. Após demonstrar as falhas de otimização, é proposta uma nova *stack* TCP/IP, a fim de sanar estes problemas de performance. De acordo com (JEONG *et al.*, 2014) alguns dos principais problemas da atual *stack* TCP/IP Linux são:

1. **Falta de localidade de conexão:** Grande maioria das aplicações hoje em dia são *multi-threaded* a fim de tirar melhor proveito de sistemas multinúcleos. Entretanto, *sockets* de escuta para conexões são normalmente compartilhados entre os núcleos, criando a necessidade de uma trava para garantir o acesso correto a fila de aceitação de conexões TCP/IP entre os múltiplos núcleos, causando latência no processamento da conexão. Além disso, o núcleo que executa o código de alto nível para processamento TCP/IP dentro do *kernel* não é necessariamente o mesmo que executará o código a nível de Aplicação que recebe e envia dados por meio do *socket*, possivelmente causando mais faltas de cache que também prejudicam a performance.

2. **Espaço de descritores de arquivos compartilhado:** Em sistemas operacionais que seguem a interface POSIX (GROUP, 2023), o espaço dos descritores de arquivos dentro de um processo é compartilhado. No caso do Linux, quando um novo *socket* será alocado, é feita uma busca pelo menor valor de descritor de arquivo. Conforme apresentado em (BOYD-WICKIZER *et al.*, 2008), em cenários de tráfego intenso nos quais existem muitas conexões concorrentes, essa busca por diferentes *threads* por um descritor de arquivo causa um gargalo significativo devido ao número de faltas de cache e a travas de acesso concorrente a esta estrutura de dados compartilhada entre as *threads*

Além destes gargalos, é também salientada a degradação de performance devido a chamadas de sistemas excessivas e processamento por pacote, ambos assuntos já discutidos anteriormente neste trabalho.

Os autores então descrevem as decisões de projeto feitas para a biblioteca mTCP, tendo em vista diversas estratégias para evitar os gargalos de performance comentados anteriormente neste trabalho. Por exemplo, esta nova *stack* TCP/IP implementa todo o tráfego de dados em espaço de usuário, assim, eliminando chamadas de sistemas excessivas devido a necessidade do uso do *kernel*. Outro exemplo de estratégia otimização é a implementação de uma *thread* processadora por núcleo de processamento, ou seja, cada conexão TCP/IP será executada do começo ao fim por um mesmo núcleo, favorecendo a localidade de cache dos processadores, e amortizando a execução de protocolos de coerência de cache, já que não haverá escritas e leituras de processadores interferindo umas com as outras. Além disso, utilizando estruturas de dados "amigáveis" à cache e *Lock-free* (LANGDALE, 2023), que eliminam a espera em uma trava de acesso a estruturas de dados compartilhadas, e também reduzem faltas na TLB, entre outras estratégias diversas.

O design da API mTCP (JEONG *et al.*, 2014) é feita inspirada nos *sockets* BSD (WIKIPEDIA, 2023), com o intuito de agilizar possíveis migrações de sistemas já existentes e facilitar a criação de novos sistemas. Ao final do trabalho, é demonstrada que a portabilidade de sistemas já existentes oferece pouco atrito de implementação devido a API similar a disponibilizada pelo sistema operacional Linux, e podendo alcançar uma melhoria significativa em performance, como foi o caso da migração do servidor web lighttpd que obteve aumento de vazão de 2.2x quando comparado com a implementação tradicional.

4 DESENVOLVENDO APLICAÇÕES COM MTCP E DPDK

Neste capítulo serão descritos alguns dos processos e pré-requisitos necessários para desenvolver corretamente aplicações distribuídas utilizando em conjunto as bibliotecas mTCP e DPDK. Além disso também será comentado um pouco sobre a implementação da biblioteca mTCP e suas especificidades.

4.1 PRÉ-REQUISITOS

Antes de propriamente desenvolver utilizando ambas as bibliotecas, existem alguns pré-requisitos para que as bibliotecas de Kernel-Bypass funcionem corretamente e operem no seu maior desempenho.

4.1.1 Compatibilidade de Hardware/Software

Para desenvolver uma aplicação que utilize a biblioteca DPDK deve-se possuir *hardware* compatível. Componentes como placas de rede, processadores, motores de DMA e motores criptográficos precisam estar de acordo com os suportados pela biblioteca. Por ser uma iniciativa da Intel, inicialmente a biblioteca era suportada apenas por placas de rede e processadores da empresa, entretanto, com o passar do tempo o suporte foi adicionado para mais dispositivos. Atualmente, na parte de arquitetura de processadores, além da arquitetura x86 (originária do microprocessador Intel 8086), são também suportadas Advanced RISC Machines (ARM) e Power PC (PPC); na parte de placas de rede, além de obviamente suportar placas Intel, são também suportadas placas de rede NVIDIA, Marvell, Advanced Micro Devices (AMD) e até mesmo alguns motores de virtualização, como por exemplo o Google Cloud VMware Engine (GVE), entre outras. A lista completa de *hardware* compatível com a biblioteca pode ser acessada pelo endereço `http://core.dpdk.org/supported/`.

Além dos pré-requisitos de *hardware*, precisa-se também garantir algumas condições a nível de *software*. Para o correto funcionamento da biblioteca mTCP é necessário utilizar uma versão mais antiga do GNU Compiler Collection (GCC) do que as disponibilizadas pelas versões mais atuais dos sistemas operacionais Unix. Além disso, a versão do *kernel* Linux usada também precisa ser mais antiga. Nestes experimentos, foram utilizados a versão do *kernel* 4.15.0-55-generic e a versão do GCC 7.4.0.

4.1.2 Configuração do Sistema e Bibliotecas

Aqui serão descritos todos os passos necessários para configurar corretamente as bibliotecas de Kernel-Bypass, assim como etapas necessárias para configurar o sistema operacional para o melhor funcionamento com ambas as bibliotecas.

4.1.2.1 DPDK

Primeiramente é necessário realizar a alocação de *Hugepages*. Para estes experimentos foram alocadas 8192 páginas de 2048KB para cada processador Non-Uniform Memory Access (NUMA), totalizando 32GB de memória alocada, por meio dos comandos:

```
echo 8192 > /sys/devices/system/node/node0/hugepages/hugepages-2048kB/nr_hugepages
echo 8192 > /sys/devices/system/node/node1/hugepages/hugepages-2048kB/nr_hugepages
```

Listagem 4.1 – Configuração das *Hugepages*

Além da alocação das *Hugepages*, conforme descrito na Seção 2.4, a biblioteca DPDK utiliza os *Poll Mode Drivers* (PMDs) como os *drivers* para a placa de rede, por isso, é necessário associar a placa de rede que será utilizada pela aplicação com os *drivers* disponibilizados pela biblioteca. Importante ressaltar que após associada com este *driver* a placa de rede ficará indisponível para o sistema operacional, podendo ser utilizada apenas por aplicações que utilizem a biblioteca DPDK. Além disso, para garantir que as duas máquinas conectadas em Local Area Network (LAN) consigam se comunicar por meio de aplicações DPDK é necessário que as placas de rede associadas com este *driver* estejam previamente conectadas diretamente em ambas as máquinas.

4.1.2.2 mTCP

A primeira coisa a se fazer após clonar o repositório do projeto mTCP é inicializar os submodulos disponibilizados pela ferramenta `git` por meio dos comandos mostrados na Listagem 4.2, é desta forma que a biblioteca tem acesso ao código referente a biblioteca DPDK. Após baixar o código necessário, a biblioteca mTCP possui um programa interativo de configuração `./setup_mtcp_dpdk_env.sh` que permite realizar os passos descritos na seção 4.1.2.1. Ao final da execução do programa é possível configurar uma interface de rede, utilizando o comando listado em 4.3 para monitorar estatísticas da placa de rede associada com os *drivers* PMD.

```
git submodule init
git submodule update
```

Listagem 4.2 – Subindo a interface de rede para visualização de estatísticas

```
sudo ifconfig dpdk0 x.x.x.x netmask 255.255.255.0 up
```

Listagem 4.3 – Subindo a interface de rede para visualização de estatísticas

Após as configurações prévias, é necessário executar o *script* de configuração listado em 4.4, no qual `$RTE_SDK` é o caminho para a biblioteca DPDK previamente baixada com o repositório mTCP, e `$RTE_TARGET` é a aplicação alvo para qual que queremos configurar a biblioteca, no nosso caso este valor é `x86_64-native-linuxapp-gcc`. Por fim, pode-se compilar e instalar a biblioteca por meio do comando `make`. A partir deste momento a

biblioteca poderá ser utilizada por todo o sistema como uma biblioteca na linguagem de programação C qualquer.

```
./configure --with-dpdk-lib=$RTE_SDK/$RTE_TARGET
```

Listagem 4.4 – Script de configuração da biblioteca mTCP

Mais informações sobre a configuração da biblioteca podem ser achadas no repositório oficial do projeto `https://github.com/mtcp-stack/mtcp`.

4.2 IMPLEMENTAÇÃO

Nesta seção serão descritos detalhes de implementação e uso da biblioteca mTCP, assim como um pouco do funcionamento interno desta e como explorar o máximo de desempenho possível.

4.2.1 Inicialização da aplicação e arquivo de configuração

Todas as aplicações mTCP necessitam de um arquivo de configuração que será utilizado na inicialização da aplicação. Este arquivo contém informações como o número de núcleos que será disponibilizado para nossa aplicação mTCP, o número máximo de conexões por núcleo, tamanho dos *buffers* de recebimento e envio dos *sockets* mTCP, etc. O carregamento deste arquivo de configuração é o primeiro passo em qualquer aplicação mTCP e deve ser carregado por meio da chamada de função na primeira linha da Listagem 4.5.

```
1  mtcp_init(conf_file);
2  mctx_t mctx = mtcp_create_context(core);
3
4  /* Cofigurando socket do servidor */
5  int mtcp_listener = mtcp_socket(mctx, AF_INET, SOCK_STREAM, 0);
6  mtcp_setsock_nonblock(mctx, mtcp_listener); // Socket nao bloqueante
7  mtcp_bind(mctx, mtcp_listener,
8          (struct sockaddr *)&server_address, sizeof(struct sockaddr_in));
9  mtcp_listen(mctx, mtcp_listener, BACKLOG);
10 /* Fim da configuracao */
```

Listagem 4.5 – Inicialização e configuração básica da biblioteca mTCP

4.2.2 Criação do contexto mTCP

Conforme descrito na Seção 3, uma das decisões de projeto da biblioteca é criar uma *thread* mTCP por *thread* de aplicação. Esta *thread* mTCP é associada necessariamente a um núcleo de processamento e é responsável por realizar todo o processamento TCP/IP, como por exemplo criações de novas conexões. A *thread* de aplicação se comunica com a *thread* mTCP por meio de *buffers* compartilhados accessíveis por meio de chamadas

funções disponibilizadas pela biblioteca (e.g., `mtcp_listen()`). Quando a aplicação deseja alterar alguma informação compartilhada, como por exemplo utilizando uma chamada de escrita `mtcp_write()`, uma requisição de processamento é colocada em uma das filas de processamento mantidas pela biblioteca. Esta fila eventualmente será esvaziada e processada pela *thread* mTCP.

Para realizar a criação da *thread* mTCP a biblioteca disponibiliza a função chamada na linha 2 da Listagem 4.5, que recebe como parâmetro o identificador do processador que será responsável por processar esta *thread* mTCP. Existem duas regras para a passagem deste identificador:

- O identificador não pode ser repetido em outras chamadas desta função. Ou seja, apenas uma *thread* mTCP pode ser criada por processador;
- O identificador necessita mapear para um processador existente. Por exemplo, caso sua máquina possua 8 processadores, o valor válido para este argumento é de 0 até 7.

É importante salvar o retorno desta função em uma variável, pois o contexto criado é utilizado nas chamadas subsequentes da biblioteca.

4.2.3 Utilização da API de comunicação em rede

Após ter configurado corretamente a biblioteca e iniciado a *thread* mTCP é possível começar a comunicação distribuída. A biblioteca utiliza *sockets* não bloqueantes e uma API bem similar a API `epoll` disponibilizada pelo *kernel* Linux. A Listagem 4.6 mostra uma exemplo simplificado de como funciona um servidor Eco implementado utilizando a biblioteca. Na Listagem 4.6, das Linhas 1 até 7, é feita a inicialização da estrutura `mtcp_epoll` e a criação do *socket* servidor, que será responsável por aceitar novas conexões. A partir da Linha 8, é tem-se o *loop* de processamento eventos de E/S, é neste *loop* que serão verificadas se as requisições de processamento são do tipo leitura, nova conexão ou erro. Caso sejam requisições de novas conexões, utiliza-se a chamada `mtcp_accept(mctx, listener, NULL, NULL)` da Linha 16, o *socket* é configurado para ser não bloqueante e, por fim, o novo *socket* cliente é adicionado na lista de *sockets* monitorados. Caso a requisição seja de leitura de dados, o servidor realiza a leitura com a chamada da Linha 23 e, logo em seguida, escreve o valor lido de volta no *socket* cliente com a invocação da função na Linha 28.

```
1  char recv_buff[BUFF_SIZE];
2  int ep_id = mtcp_epoll_create(mctx, MAX_EVENTS);
3  struct mtcp_epoll_event ev, events[MAX_EVENTS];
4  ev.events = MTCP_EPOLLIN;
5  ev.data.sockid = listener;
6
7  mtcp_epoll_ctl(mctx, ep_id, MTCP_EPOLL_CTL_ADD, listener, &ev);
8  while(!done) {
9      count = mtcp_epoll_wait(mctx, ep_id, events, MAX_EVENTS,
       EPOLL_TIMEOUT);
10     for(int i = 0; i < count; i++) {
11         event = events[i];
12         int sockid = event.sock.id
13
14         /* Requisicao de nova conexao */
15         if(sockid == listener) {
16             int client = mtcp_accept(mctx, listener, NULL, NULL);
17             ev.events = MTCP_EPOLLIN;
18             ev.data.sockid = client;
19
20             mtcp_setsock_nonblock(mctx, client);
21             mtcp_epoll_ctl(mctx, ep_id, MTCP_EPOLL_CTL_ADD, client, &ev)
       ;
22         } else if(event.events & MTCP_EPOLLIN) {
23             int read = mtcp_read(mctx, sockid, recv_buff, BUFF_SIZE);
24             if(read <= 0) {
25                 mtcp_close_connection(mctx, sockid, ep_id);
26                 continue;
27             }
28             mtcp_write(mctx, sockid, recv_buff, BUFF_SIZE);
29         }
30         else if((event.events & MTCP_EPOLLERR) || (event.events &
       MTCP_EPOLLHUP)) {
31             if(errno != EAGAIN) {
32                 mtcp_close_connection(ctx, sockid, ep_id);
33             }
34         }
35     }
36 }
```

Listagem 4.6 – Exemplo de um servidor escrito a partir da biblioteca mTCP

Ainda na Listagem 4.6, pode-ser perceber que as invocações das funções da biblioteca se assemelham bastante as chamadas de sistema disponibilizadas pelo sistema operacional Linux. Por exemplo a chamada de sistema `write(fd, send_buffer, BUFFER_SIZE)` traduz para a invocação da função `mtcp_write(mctx, fd, send_buffer, BUFFER_SIZE)` (Linha 27 da Listagem 4.6), na qual o parâmetro `mctx` é o valor retornado pela função

de criação de contexto listada em 4.5. Assim, é seguro assumir que um desenvolvedor com conhecimento prévio da API de processamento de eventos disponibilizada pelo Linux não deve ter grandes dificuldades no desenvolvimento com a biblioteca, já que seus comportamentos se assemelham bastante.

5 AVALIAÇÃO EXPERIMENTAL

Este capítulo apresenta a comparação de desempenho de duas aplicações Cliente-Servidor; uma utilizando estratégias tradicionais de comunicação de E/S, sendo o fluxo de dados trafegado pelo *kernel* e a outra construída a partir de duas bibliotecas de Kernel-Bypass, fazendo com que o fluxo de dados entre os nodos da aplicação seja trafegado totalmente pelo espaço de usuário, sem envolvimento do *kernel*. Foram criados três cenários para avaliar o desempenho das bibliotecas de Kernel-Bypass em comparação com a pilha de rede do sistema operacional Linux em diferentes situações. Para fins desta deste trabalho, a palavra "requisições" é utilizada para se referir ao envio de uma mensagem para o servidor e sua devida resposta sendo enviada de volta para o cliente, assim, constituindo uma requisição servida. No cenário 1 é explorada a capacidade de escalabilidade no número requisições servidas pelo servidor, aumentando gradativamente o número de servidores disponíveis, dado que a quantidade de conexões clientes está saturada. Este cenário tem como propósito analisar o desempenho de ambas bibliotecas em uma situação de alta concorrência e a escalabilidade no número de requisições servidas, dado o aumento no número de *threads* de servidores. No segundo cenário foi analisada a capacidade de ambas aplicações saturarem a largura de banbda da placa de rede, neste cenário foram estabelecidos casos de teste com duas e quatro *threads* de servidor, variando o tamanho da mensagem enviadas e mantendo um número fixo de conexões clientes. Já no terceiro cenário foi explorado um caso de uso mais comum para aplicações distribuídas, sem necessariamente saturar o número de clientes e levar o servidor ao estresse máximo. Neste cenário manteve-se apenas um servidor e foi-se gradualmente aumentando o número de *threads* clientes, mantendo apenas uma conexão por *thread*, para analisar a relação entre o número de requisições servidas e latência em ambas aplicações.

5.1 BANCADA DE TESTES

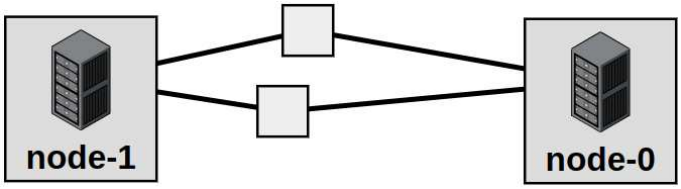
Como plataforma de *testbed* foi utilizado o ambiente de testes distribuídos Emulab (WHITE *et al.*, 2002), que disponibiliza infraestrutura sob demanda para execução de experimentos.

Nestes experimentos, foram utilizados nodos denominados d430, compostos por máquinas que possuem dois processadores E5-2630 operando na frequência 2.4GHz, cada um destes processadores equipado 8 núcleos, com 2 *threads* por núcleo, totalizando 16 *threads* por processador. Além disso, possuindo 64GB de memória RAM dividida entre 8 pentes de 8GB e placas de rede com velocidade de transmissão de 10Gb/s. Mais informações sobre este tipo de nodo podem ser encontradas por meio do endereço `https://gitlab.flux.utah.edu/emulab/emulab-devel/-/wikis/Utah-Cluster/d430s`

Em relação à infraestrutura do experimento, utilizou-se duas máquinas conectadas em LAN contando com duas conexões com velocidade de transmissão de 10Gb/s, conforme

mostrado na Figura 8. Ambas as máquinas utilizaram o sistema operacional Ubuntu Linux 18.04.

Figura 8 – Topologia da rede utilizada para os experimentos.

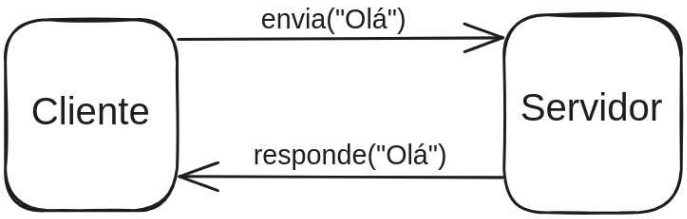


Fonte: Elaborado pelo autor (2023)

5.2 ARQUITETURA DA APLICAÇÃO DE TESTE

Para comparar as estratégias de E/S distribuídas, foram implementados duas aplicações tipo Eco. Conforme demonstrado na Figura 9, neste tipo de aplicação o cliente envia uma mensagem para o servidor e este responde a mesma mensagem de volta para o cliente. Para cada uma destas aplicações foram implementados um servidor e um cliente que realizam sua troca de mensagens por meio do protocolo de comunicação TCP. Em um dos protótipos, tanto o cliente quanto o servidor foram implementados a partir das bibliotecas mTCP e DPDK com estratégias de Kernel-Bypass, desviando o fluxo de troca de dados do *kernel* e realizando-o totalmente em espaço de usuário; enquanto isso o outro protótipo utiliza a API de comunicação de rede nativa do *kernel* Linux. As Figuras 10 e 11 ajudam a visualizar como é feita a troca de mensagens entre os protótipos que utilizam de Kernel-Bypass e os que se comunicam de forma tradicional, respectivamente.

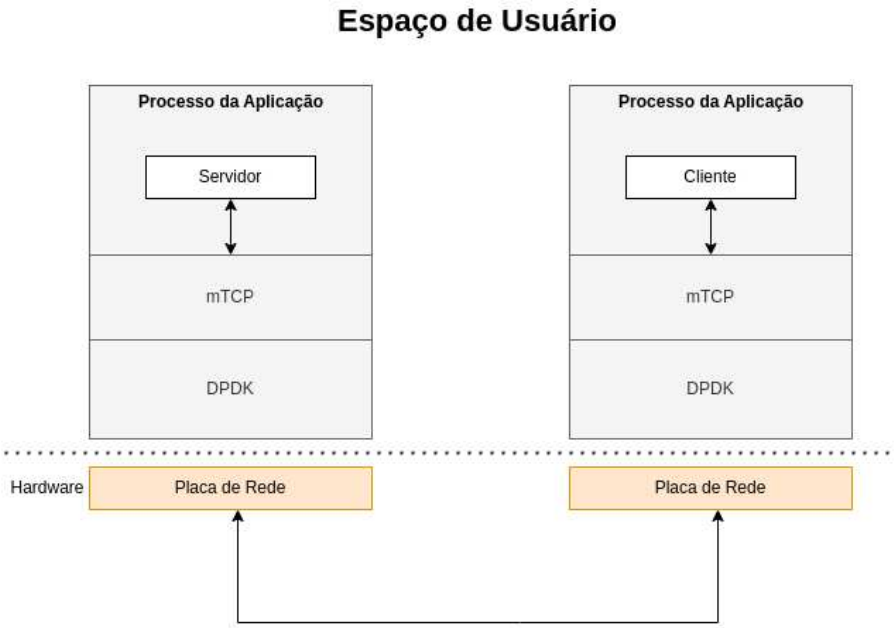
Figura 9 – Arquitetura de um sistema Eco.



Fonte: Elaborado pelo autor (2023)

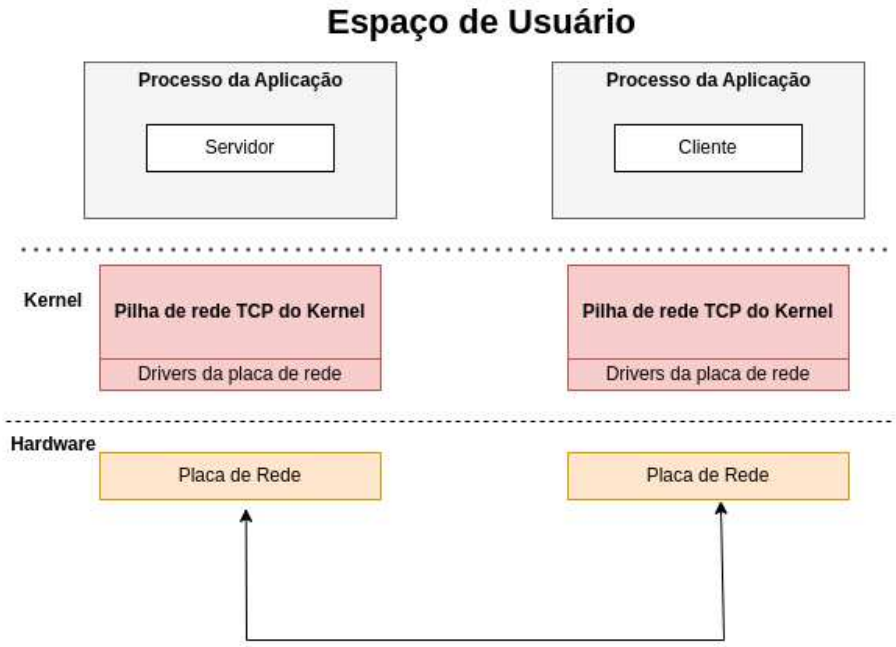
Ambos protótipos possuem conexões persistentes, ou seja, após estabelecida a conexão entre cliente e servidor, a mesma só é interrompida pelo fim da execução do processo. Além disso, a troca de dados entre cliente e servidor é contínua, o fluxo de envio e resposta do servidor eco é repetido até o fim do processo.

Figura 10 – Demonstração do fluxo de troca de dados utilizando as estratégias de Kernel-Bypass.



Fonte: Elaborado pelo autor (2023)

Figura 11 – Demonstração do fluxo de troca de dados utilizando a API de rede nativa do Linux.



Fonte: Elaborado pelo autor (2023)

5.3 AVALIAÇÃO

Visando explorar o desempenho das estratégias de Kernel-Bypass em diferentes casos de uso foram explorados três cenários: o primeiro simulando um cenário de alta

concorrência e intenso fluxo de dados, no qual é analisada a capacidade da aplicação de lidar com um grande número de conexões simultâneas e como a o número de requisições servidas por esta aplicação escala com a quantidade de servidores disponíveis; o segundo cenário explora a capacidade dos protótipos de saturar a largura de banda da placa de rede com diferentes tamanhos de mensagens enviados e o terceiro cenário simula um cenário de baixa concorrência e baixa demanda dos servidores, analisando o desempenho das aplicações em casos onde o fluxo de troca de dados não é tão intenso.

5.3.1 Cenário 1

Neste cenário foi explorada a capacidade dos protótipos para alta concorrência na troca de mensagens pequenas. O objetivo é analisar a capacidade de escalabilidade de servir requisições TCP dado um cenário saturado de conexões clientes, variando apenas o número de *threads* de servidores. Nestes experimentos, foram utilizadas 8 *threads* clientes, cada uma destas *threads* criando 2000 conexões persistentes com os servidores disponíveis. Após estabelecer a conexão, foi medido o número de requisições servidas pelo servidor em um período de 60 segundos. Este cenário foi divido em dois casos de teste, um enviando mensagens com 64 bytes de tamanho e o outro utilizando mensagens de 128 bytes.

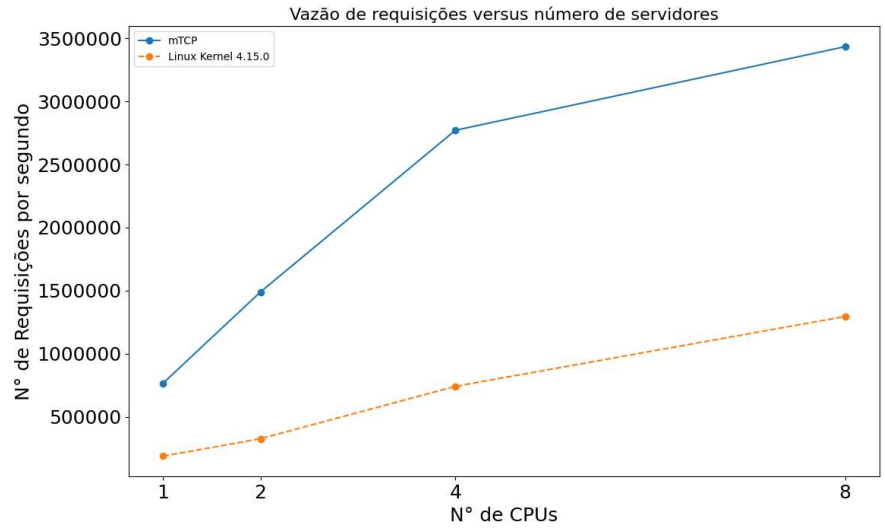
A Figura 12 mostra a quantidade total de requisições que foi servida por segundo por cada uma das implementações do protótipo, utilizando tamanhos de mensagens de 64 bytes e 128 bytes. Pode-se perceber que a solução Kernel-Bypass demonstrou grande eficiência no processamento de pacotes de pequeno tamanho, não só conseguindo servir um número maior de requisições com apenas uma *thread* de servidor do que a solução tradicional, mas também possuindo um comportamento linear quando incrementa-se em até 4 o número de *threads* de servidores, a partir de qual o comportamento torna-se menos linear. Enquanto isso, mesmo com a solução tradicional utilizando o *kernel* Linux oferecendo certo grau de escalabilidade, a performance não se compara com a solução Kernel-Bypass, que ofereceu 265% de melhoria no pior caso e 458% no melhor caso.

5.3.2 Cenário 2

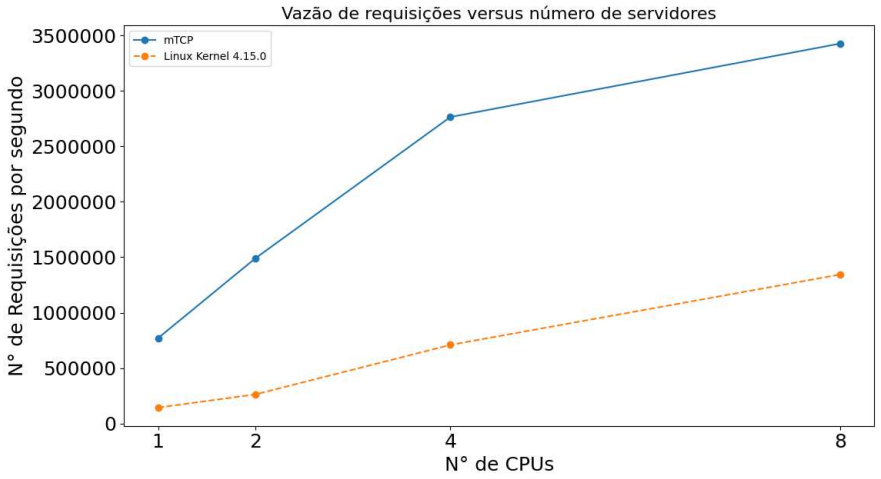
Neste cenário foi analisado o potencial de ambos protótipos saturarem a largura de banda da placa de rede. Para isto, foram criados dois casos de teste; um deles utilizando duas *threads* de servidores e o outro utilizando 4 *threads* de servidores, em ambos casos de teste foram utilizados 4 *threads* de clientes, cada cliente contando com 3000 conexões persistentes e mantendo o comportamento da aplicação conforme descrito na Seção 5.2. Após estabelecidas todas as conexões foi monitorada a taxa de transmissão da placa de rede utilizada.

Analisando os resultados apresentados na Figura 13 pode-se perceber que os casos de teste desenvolvidos a partir das bibliotecas mTCP e DPDK possuem a capacidade de saturar a placa de rede em alguns casos. Enquanto isso, é notável que em nenhum dos

Figura 12 – Resultados da execução do Cenário 1.



(a) Resultados da execução do Cenário um com tamanho de mensagem de 64 bytes



(b) Resultados da execução do Cenário um com tamanho de mensagem de 128 bytes

Fonte: Elaborado pelo autor (2023)

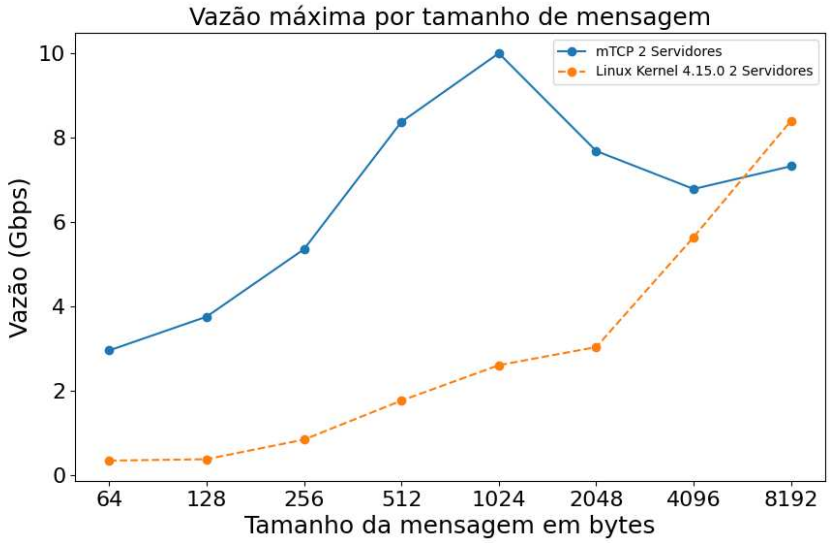
casos com a pilha de rede do *kernel* Linux a capacidade máxima da placa de rede foi alcançada, tendo sua vazão máxima de 8.41Gb/s em pacotes de 8192 bytes de tamanho e com duas *threads* de servidores. Percebe-se, na Figura 13a, que o caso de testes que utiliza 2 *threads* de servidores do protótipo Kernel-Bypass tem sua performance máxima com mensagens de 1024 bytes, experimentando degradação em vazão com valores de mensagens maiores. Estipula-se que esta queda possa estar associada com as configurações de Maximum Transmission Unit (MTU) da placa de rede e a capacidade da biblioteca de lidar com a fragmentação de pacotes. Entretanto, ao analisar o segundo caso de teste, apresentado pela Figura 13b, ao dobrar o número de servidores disponíveis, após a vazão máxima ser alcançada com mensagens de 1024 bytes de tamanho, a placa se manteve saturada com tamanhos de pacote maiores, alcançando uma taxa de transmissão de 10Gb/s. Aprofundando a análise da Figura 13b, vemos que, diferente da implementação Kernel-Bypass, ao dobrar o número de servidores disponíveis na solução tradicional, não temos um aumento na vazão máxima do protótipo, e sim o efeito contrário, baixando a vazão máxima para 5.72Gb/s em casos de mensagens com tamanho de 4096 bytes, representando uma queda de quase 30% do caso de teste com 2 servidores.

5.3.3 Cenário 3

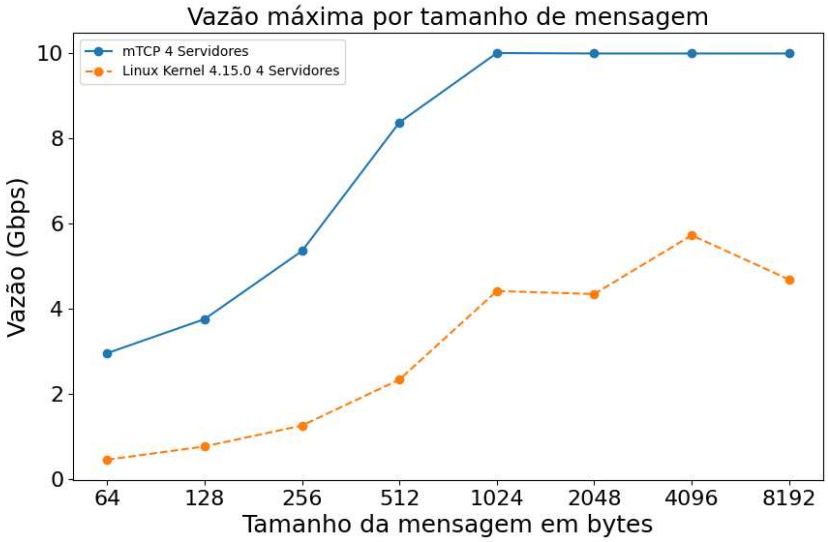
Neste cenário optou-se por explorar um caso mais comum de desenvolvimento de aplicações distribuídas, sem necessariamente saturar a quantidade de conexões de clientes. Optou-se por fixar apenas um servidor e gradativamente aumentar o número de *threads* clientes, entretanto, ao invés de cada *thread* cliente possuir diversas conexões como nos exemplos anteriores, cada *thread* cliente contou com apenas uma conexão, cada uma destas conexões enviando mensagens de 2048 bytes. Para este cenário, foram utilizadas respectivamente 1, 2, 4, 8 e 16 *threads* clientes, coincidindo com os pontos no eixo X da Figura 14 de cada um dos protótipos desenvolvido. No caso do protótipo de Kernel-Bypass, a cada nova *thread* cliente foi também criada uma nova *thread* mTCP. Após encerrar o experimento, foi utilizado o percentil 95 das latências medidas pelos protótipos para a análise de resultados. Analisando a Figura 14 pode-se perceber que a biblioteca mTCP teve um desempenho inferior quando comparada a pilha de rede Linux, acarretando um aumento de latência de diminuição na vazão de requisições servidas. Enquanto isso, o protótipo que utiliza estratégias tradicionais de E/S teve impacto mínimo na sua latência conforme a vazão aumentava devido ao aumento no número de *threads* clientes, a aplicação desenvolvida a partir das bibliotecas mTCP e DPDK demonstrou degradação significativa na latência, alcançando um aumento de 18x no pior caso.

Estipula-se que esta queda em performance deve-se ao custo operacional acarretado para manter uma *thread* mTCP; conforme descrito na Seção 4.2.2 cada uma destas *threads* é responsável por realizar todo o fluxo de processamento TCP/IP, gerenciando trocas de mensagens entre a *thread* de aplicação e a placa de rede. Conforme viu-se no Cenário 1, a

Figura 13 – Resultados da execução do Cenário 2 utilizando Kernel-Bypass.



(a) Execução do segundo cenário utilizando 2 servidores

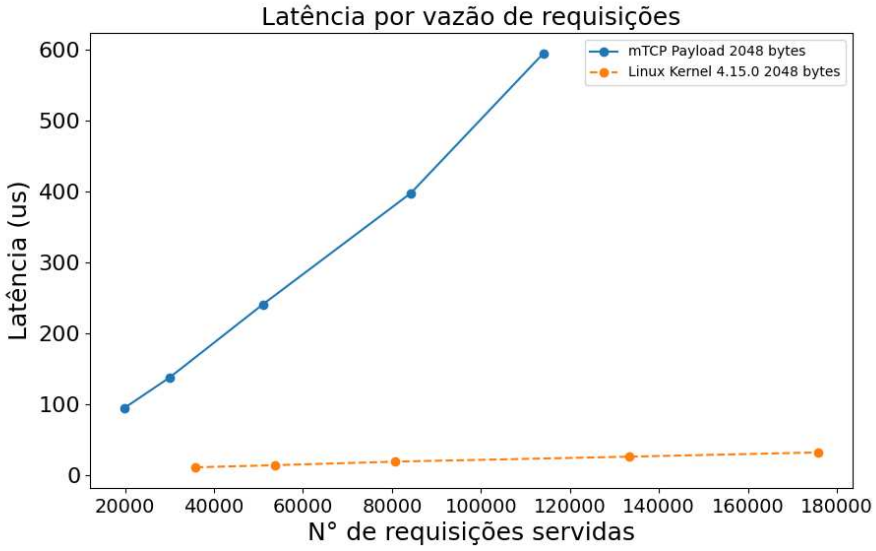


(b) Execução do segundo cenário utilizando 4 servidores

Fonte: Elaborado pelo autor (2023)

biblioteca mTCP possui grande potencial quando criamos um grande número de conexões por *thread* mTCP, entretanto, ao criar apenas uma conexão nesta *thread* estamos lidando com todo o custo operacional causado pela mesma sem poder tirar total proveito de suas vantagens em casos de alta concorrência.

Figura 14 – Gráfico de latência versus vazão de requisições para mensagens com 2048 bytes de tamanho.



Fonte: Elaborado pelo autor (2023)

6 CONCLUSÃO

Neste trabalho buscou-se estudar e apresentar os gargalos conhecidos no processamento de rede oferecido pelo *kernel* Linux ao mesmo tempo em que buscava-se uma alternativa mais performática. Ao aprofundar os estudos nas estratégias de Kernel-Bypass, notou-se que muito dos gargalos conhecidos poderiam ser resolvidos por meio da implementação de funcionalidades tradicionalmente do *kernel*, como por exemplo processamento TCP/IP, em espaço de usuário na forma de bibliotecas.

A fim de validar o desempenho das estratégias de Kernel-Bypass foram escolhidas as bibliotecas DPDK e mTCP, responsáveis respectivamente por processamento de pacotes e processamento TCP/IP, para desenvolver um protótipo que trafega dados entre aplicações distribuídas completamente pelo espaço de usuário, sem interferência do *kernel* na troca de informação. A partir da implementação deste protótipo uma série de testes em diferentes cenários foi realizada para analisar métricas como vazão e latência e compará-las a um protótipo desenvolvido utilizando a API de rede tradicional do Linux, ou seja, trafegando os dados pelo *kernel*.

Por meio dos cenários de teste provou-se que as bibliotecas DPDK e mTCP possuem escalabilidade e desempenho em vazão superiores a pilha de rede do *kernel* Linux quando realiza-se processamento de pacotes pequenos e com um grande número de conexões persistentes. Além disso, o protótipo Kernel-Bypass mostrou-se capaz de saturar a largura de banda da placa de rede, enquanto a aplicação desenvolvida utilizando *kernel* Linux não foi capaz do mesmo feito. Entretanto, em cenários de pouca concorrência a aplicação Kernel-Bypass foi inferior na relação latência x vazão quando comparada a outra implementação, esta degradação em performance apenas salienta que este tipo de estratégia tem como foco ser aplicada em aplicações que antecipam alta vazão de dados e um grau alto de concorrência, não sendo adequadas para aplicações que não tem como objetivo um intenso fluxo de dados.

6.1 TRABALHOS FUTUROS

Mesmo este trabalho tendo explorado de forma teórica algumas estratégias de Kernel-Bypass a avaliação experimental foi limitada a apenas um conjunto de duas bibliotecas específicas. Esta limitação abre portas para a exploração de mais implementações de bibliotecas e dispositivos Kernel-Bypass disponíveis. Um exemplo seria aprofundar o conhecimento em dispositivos RDMA, que permitem acesso à memória de um computador para a memória de outro computador sem envolver o sistema operacional de nenhum dos computadores, este tipo de técnica é dependente de placas de rede específicas. Ou então a biblioteca F-Stack (JIANG, 2023), que também oferece uma implementação dos protocolos TCP/IP similar a mTCP e totalmente a nível de usuário, sem interferência do *kernel*.

Além disso, pensando nas bibliotecas mTCP e DPDK, poderiam ser feitas migrações de sistemas existentes que atualmente utilizam como camada de rede o *kernel* Linux para utilizarem estas bibliotecas, a fim de medir o impacto do uso dessas estratégias em um sistema real. Um exemplo seria a biblioteca `librdkafka` (EDENHILL, 2023), que implementa o protocolo Apache Kafka de Produtores/Consumidores na linguagem de programação C. Ainda tendo as bibliotecas mTCP e DPDK em mente, seria possível explorar placas de rede com uma largura de banda maior para obter uma noção exata do limite da taxa de transferência de dados do protótipo teste. Conforme observado nos testes, a aplicação exemplo desenvolvida foi capaz de alcançar a taxa de vazão de 10Gbp/s, sendo o único limitante o próprio limite de largura de banda da placa.

Outra abordagem interessante, mantendo-se coerente a proposta de estratégias para otimização de operações de E/S em sistemas distribuídos de alta vazão, poderia ser a investigação de algoritmos com tratamento eficiente para operações de E/S, como por exemplo estudo de algoritmos *cache-oblivious* que buscam tirar melhor proveito da memória cache sem ter como pré-requisito a configuração da mesma. Ou ainda explorar o uso de estruturas de dados eficientes para operações de E/S, como a árvores de *buffers* (ARGE, 2003), um tipo especial de árvore binária que possui operações otimizadas para E/S.

Por fim, tendo em vista as ineficiências no processamento de rede do *kernel* Linux apresentadas neste trabalho, outra abordagem válida seria estudar mais a fundo o *kernel* Linux e propor possíveis alterações no projeto do atual deste a fim de sanar os gargalos encontrados.

REFERÊNCIAS

ALAM, Tanweer. Cloud Computing and Its Role in the Information Technology. **SSRN Electronic Journal**, jan. 2020. DOI: 10.2139/ssrn.3639063.

ARGE, Lars. The Buffer Tree: A Technique for Designing Batched External Data Structures. **Algorithmica**, Springer-Verlag, Berlin, Heidelberg, v. 37, n. 1, p. 1–24, set. 2003. ISSN 0178-4617. DOI: 10.1007/s00453-003-1021-x. Disponível em: https://doi.org/10.1007/s00453-003-1021-x.

ASSUNCAO, Marcos *et al.* Big Data computing and clouds: Trends and future directions. **Journal of Parallel and Distributed Computing**, v. 75, jan. 2014.

BARROSO, Luiz *et al.* Attack of the Killer Microseconds. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 60, n. 4, p. 48–54, mar. 2017. ISSN 0001-0782. DOI: 10.1145/3015146. Disponível em: https://doi.org/10.1145/3015146.

BELKHIRI, Adel *et al.* Performance analysis of DPDK-based applications through tracing. **Journal of Parallel and Distributed Computing**, v. 173, p. 1–19, 2023. ISSN 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2022.10.012. Disponível em: https://www.sciencedirect.com/science/article/pii/S0743731522002271.

BOYD-WICKIZER, Silas *et al.* Corey: An Operating System for Many Cores. *In:* PROCEEDINGS of the 8th USENIX Conference on Operating Systems Design and Implementation. San Diego, California: USENIX Association, 2008. (OSDI'08), p. 43–57.

CHEN, Ruining; SUN, Guoao. A Survey of Kernel-Bypass Techniques in Network Stack. *In:* PROCEEDINGS of the 2018 2nd International Conference on Computer Science and Artificial Intelligence. Shenzhen, China: Association for Computing Machinery, 2018. (CSAI '18), p. 474–477. DOI: 10.1145/3297156.3297242. Disponível em: https://doi.org/10.1145/3297156.3297242.

CISCO. **Cisco Annual Internet Report (2018–2023) White Paper**. Disponível em: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html. Acesso em: 14 jun. 2023.

COX, Alan. **Network Buffers and Memory Management**. [*S.l.*: *s.n.*], 1996. Disponível em: https://www.linuxjournal.com/article/1312. Acesso em: 29 mai. 2023.

EDENHILL, Magnus. **librdkafka - the Apache Kafka C/C++ client library**. Disponível em: https://github.com/confluentinc/librdkafka. Acesso em: 13 nov. 2023.

FIRESTONE, Daniel *et al.* Azure Accelerated Networking: SmartNICs in the Public Cloud. *In:* 15TH USENIX Symposium on Networked Systems Design and Implementation (NSDI 18). Renton, WA: USENIX Association, abr. 2018. P. 51–66. Disponível em: https://www.usenix.org/conference/nsdi18/presentation/firestone.

GROUP, The Open. **The Single UNIX Specification Version 3 - Overview**. Disponível em: https://unix.org/version3/overview.html. Acesso em: 13 nov. 2023.

HARVEY, A. F. **DMA Fundamentals on Various PC Platforms**. Disponível em: https://cires1.colorado.edu/jimenez-group/QAMSResources/Docs/DMAFundamentals.pdf. Acesso em: 13 jun. 2023.

IETF. **RDMA Protocol Verbs Specification**. [*S.l.*: *s.n.*], 2003. Disponível em: https://www.ietf.org/archive/id/draft-hilland-rddp-verbs-00.pdf. Acesso em: 26 mai. 2023.

INTEL. **Data Plane Development Kit**. Disponível em: https://www.dpdk.org/. Acesso em: 23 mai. 2023.

JEONG, Eun Young *et al.* MTCP: A Highly Scalable User-Level TCP Stack for Multicore Systems. *In:* PROCEEDINGS of the 11th USENIX Conference on Networked Systems Design and Implementation. Seattle, WA: USENIX Association, 2014. (NSDI'14), p. 489–502.

JIANG, John. **F-Stack**. Disponível em: https://github.com/F-Stack/f-stack. Acesso em: 13 nov. 2023.

LANGDALE, Geoff. **Lock-Free Programming**. Disponível em: https://www.cs.cmu.edu/~410-s05/lectures/L31_LockFree.pdf. Acesso em: 26 mai. 2023.

LOUKIDES, Mike. **The Cloud in 2021: Adoption Continues**. Disponível em: hhttps://www.oreilly.com/radar/the-cloud-in-2021-adoption-continues/. Acesso em: 14 nov. 2023.

LOWE, David. Web System Requirements: An Overview. **Requir. Eng.**, Springer-Verlag, Berlin, Heidelberg, v. 8, n. 2, p. 102–113, jul. 2003. ISSN 0947-3602. DOI: 10.1007/s00766-002-0153-x. Disponível em: https://doi.org/10.1007/s00766-002-0153-x.

OSBORNE, Adam. **Introductions to Microcomputers: Volume One, Basic Concepts**. [*S.l.*]: McGraw-Hill Osborne Media, 1980. ISBN 0931988020.

PAGE, Linux MAN. ***sendfile* system call documentation**. Disponível em: `https://man7.org/linux/man-pages/man2/sendfile.2.html`. Acesso em: 25 mai. 2023.

PESTEREV, Aleksey *et al.* Improving Network Connection Locality on Multicore Systems. *In:* PROCEEDINGS of the 7th ACM European Conference on Computer Systems. Bern, Switzerland: Association for Computing Machinery, 2012. (EuroSys '12), p. 337–350. DOI: `10.1145/2168836.2168870`. Disponível em: `https://doi.org/10.1145/2168836.2168870`.

REDIS. **Redis Website**. Disponível em: `https://redis.io/`. Acesso em: 23 jun. 2023.

RIZZO, Luigi. Netmap: A Novel Framework for Fast Packet I/O. *In:* PROCEEDINGS of the 2012 USENIX Conference on Annual Technical Conference. Boston, MA: USENIX Association, 2012. (USENIX ATC'12), p. 9.

STEVENS, W. Richard; FENNER, Bill; RUDOFF, Andrew M. **UNIX Network Programming, Vol. 1**. 3. ed. [*S.l.*]: Pearson Education, 2003. ISBN 0131411551.

TSUNA. **How long does it take to make a context swtich?** [*S.l.*: *s.n.*], 2010. [Online; acessado em 24/05/2023]. Disponível em: `https://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html`.

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

WIKIPEDIA. **Berkley sockets**. Disponível em: `https://en.wikipedia.org/wiki/Berkeley_sockets`. Acesso em: 7 jun. 2023.

YASUKATA, Kenichi *et al.* StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs. *In:* 2016 USENIX Annual Technical Conference (USENIX ATC 16). Denver, CO: USENIX Association, jun. 2016. P. 43–56. Disponível em: `https://www.usenix.org/conference/atc16/technical-sessions/presentation/yasukata`.

ZHANG, Irene; LIU, Jing *et al.* I'm Not Dead Yet! The Role of the Operating System in a Kernel-Bypass Era. *In:* PROCEEDINGS of the Workshop on Hot Topics in Operating Systems. Bertinoro, Italy: Association for Computing Machinery, 2019. (HotOS '19), p. 73–80. DOI: `10.1145/3317550.3321422`. Disponível em: `https://doi.org/10.1145/3317550.3321422`.

ZHANG, Irene; RAYBUCK, Amanda *et al.* The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems. *In:* PROCEEDINGS of the ACM SIGOPS 28th Symposium on Operating Systems Principles. Virtual Event, Germany: Association

for Computing Machinery, 2021. (SOSP '21), p. 195–211. DOI: 10.1145/3477132.3483569. Disponível em: https://doi.org/10.1145/3477132.3483569.

APÊNDICE A – ARTIGO

Neste apêndice será apresentado o artigo no formato SBC, referente ao presente trabalho.

Estudo e implementação de estratégias para otimização de E/S em sistemas distribuídos de alta vazão, com foco em técnicas de Kernel-Bypass utilizando DPDK e mTCP

Bernardo Schmidt Farias[1], **Odorico Machado Mendizabal**[1]

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

schmidt.farias.28@gmail.com, odorico.mendizabal@ufsc.br,

Abstract. *Over the past decade, I/O devices have sped up while CPU performance has stagnated. There's a trend, especially in high data throughput environments like datacenters, that these devices are no longer the main performance bottleneck of I/O operations in high throughput distributed systems, this position is now held by the network stack processing available in Operational Systems kernel. Bearing that in mind, the discussions about techniques that bypass the kernel in the datapath processing of applications, called Kernel- Bypass techniques, are gaining more attention from researchers and companies. This kind of strategy aims to remove from the kernel the responsibility of network processing, offloading this task to libraries implemented in user space and/or hardware specific for this kind of processing. This work offers a study about the main bottlenecks in I/O processing for distributed systems made by the Linux kernel, and how the Kernel-Bypass approach optimizes these flows. In order to validate these strategies, test applications were implemented using two Kernel-Bypass libraries and its performance was compared with the linux Kernel. The results suggests that the Kernel-Bypass libraries used are able to better utilize the given hardware in some scenarios than the Linux kernel network stack. However, they're not recommended for developing systems that do not anticipate intense data flow.*

Resumo. *Na última década, os dispositivos físicos de E/S aceleraram sua velocidade de operação enquanto a performance dos processadores se manteve estagnada. Percebe-se uma tendência, principalmente em ambientes de intenso fluxo de dados como os datacenters, na qual estes dispositivos não são mais o grande gargalo de performance nas operações de E/S de sistemas distribuídos de alta vazão, mas sim o processamento de rede disponibilizado pelo sistema operacional no qual a aplicação está sendo executada. Tendo isso em vista, técnicas para desviar o processamento de dados do kernel do sistema operacional, chamadas de Kernel-Bypass, vêm conquistando interesse de pesquisadores e iniciativas privadas. Este tipo de estratégia almeja remover do kernel a responsabilidade de processamento de dados em rede, transferindo esta função para bibliotecas implementadas em espaço de usuário e/ou dispositivos físicos específicos para este fim. O presente trabalho apresenta um estudo sobre os principais gargalos do kernel Linux no processamento de E/S para sistemas distribuídos, e como as implementações de estratégias de Kernel-Bypass otimizam estes fluxos. Ao realizar a implementação de protótipos, utilizando em junção duas bibliotecas de Kernel-Bypass, foi possível avaliar o potencial*

deste tipo de estratégia quando comparado com o fluxo tradicional do kernel. Os resultados sugerem que as bibliotecas de Kernel-Bypass utilizadas conseguem extrair maior proveito do hardware disponibilizado em certos cenários quando comparadas com o kernel Linux. Entretanto, não são recomendadas para implementação de sistemas que não antecipam um fluxo intenso de troca de dados.

1. Introdução

Em 2017, o pesquisador brasileiro Luiz Barroso lançou um desafio para a comunidade científica de computação sobre otimização de operações a nível de microssegundos [Barroso et al. 2017]. Em seu artigo, é alegado que enquanto otimizações para operações em nanosegundos, como por exemplo, *branch predicition*, execução fora de ordem, e para operações de milissegundos, como troca de contexto a nível de software enquanto uma operação lenta é executada, ganham bastante atenção da comunidade científica, eventos com latência na faixa dos microssegundos têm sido deixados de lado.

Sistemas operacionais modernos necessitam de *kernels* de uso abrangente, capazes de serem executados entre diversos dispositivos; essa decisão de projeto generalista tende a deixar princípios como performance em segundo plano [Zhang et al. 2021]. Dispositivos de armazenamento e de rede de última geração já atuam na faixa de latência de um dígito de microssegundo, enquanto isso, sistemas operacionais existentes não acompanham essa velocidade. Como consequência, *kernels* tradicionais acarretam um gargalo impagável para aplicações distribuídas de alta vazão que almejam extrair total aproveitamento do *hardware* disponibilizado, como por exemplo os *datacenters* de provedores de computação em nuvem [Zhang et al. 2021]. Tendo em vista este cenário, estratégias e dispositivos que buscam desviar o fluxo de processamento de dados do *kernel* têm se tornado cada vez mais atraentes como matéria de estudo. Estes dispositivos e sistemas, popularmente conhecidos como Kernel-Bypass, ignoram o sistema operacional no fluxo de dados por meio da rede, passando responsabilidades relevantes antes cuidadas pela *stack* de rede do sistema operacional, como por exemplo recebimento e envio de pacotes TCP, para o dispositivo/sistema de Kernel-Bypass. Ao tomar como abordagem sistemas de Kernel-Bypass, permite-se que as aplicações realizem operações de E/S diretamente do espaço de usuário, ignorando operações notoriamente conhecidas por degradar a latência, como trocas de contexto e cópia dos dados entre espaço de usuário e *kernel*.

Neste trabalho foi feito um estudo sobre os principais gargalos de processamento de E/S existentes no *kernel* Linux, ao mesmo tempo em que se explorava possíveis estratégias de otimização para estas operações em sistemas distribuídos de alta vazão. Ao focar os estudos nas estratégias de Kernel-Bypass, foram escolhidas as bibliotecas mTCP e DPDK para realizar a implementação de um protótipo de sistema distribuído com o propósito de validar o potencial destas bibliotecas no aumento de performance. Ao final, os resultados indicam que o protótipo implementado utilizando as bibliotecas de Kernel-Bypass possuem uma vazão superior no processamento de pequenos pacotes em cenários de alta concorrência quando comparado com o protótipo que utiliza o processamento de rede tradicional disponibilizado pelo *kernel* Linux. Além disso, a aplicação Kernel-Bypass foi capaz de saturar a largura de banda da placa de rede utilizada, alcançando taxa de transferência de 10Gb/s, enquanto a aplicação tradicional não foi capaz do mesmo feito. Por fim, salientou-se que a aplicação desenvolvida utilizando as bibliotecas de

Kernel-Bypass não se comporta bem em cenários de baixa concorrência, oferecendo uma latência superior a aplicação tradicional. Por isso, acredita-se que o uso deste tipo de estratégia deva ser reservado para aplicações que antecipam intenso fluxo de dados e alto grau de concorrência.

2. Ineficiências do Kernel e Kernel-Bypass

As principais ineficiências do *kernel* Linux para o processamento de operações de E/S do Linux recaem em duas principais categorias: processamento de pacotes e processamento da pilha TCP/IP. A primeira categoria se refere ao processamento do pacote da placa de rede até o espaço de *kernel* e a segunda, como o nome sugere, se refere a ineficiências existentes na atual implementação do processamento TCP/IP de dentro do *kernel* Linux.

De acordo com [Chen and Sun 2018], pensando na categoria de processamento de pacotes, é possível listar três principais motivos para o sobrecusto operacional acarretado pelo *kernel*:

1. ***Overhead* de chamadas de sistema:** A cada interação entre a Aplicação e o *socket* de rede, uma chamada de sistema de sistema deve ser feita. Essas chamadas ocasionam trocas de contexto, que por si só já são custosas [Tsuna 2010], mas ainda podem ocasionar poluição da cache e expurgo da TLB, que também acabam degradando a performance;
2. **Cópias de dados:** Ao receber ou ao enviar pacotes, um pacote de rede é copiado em *buffers* tanto a nível de usuário quanto a nível de *kernel*, essas cópias tomam ciclos da CPU e degradam a performance;
3. **Processamento por pacote:** Cada pacote que chega na placa de rede é processado individualmente pelo modelo de interrupção, a placa de rede manda uma interrupção para o processador, que então processa individualmente este pacote. Como consequência de cada uma dessas interrupções, estruturas de dados pesadas como `sk_buff` [Cox 1996] tem que ser alocadas dinamicamente. Em casos nos quais muitos pacotes estão chegando para processamento, esse tipo de alocação pesada e cópia de dados pode facilmente se tornar um gargalo de processamento.

Já, quando pensa-se em ineficiências especificamente da pilha de rede TCP/IP, de acordo com [Jeong et al. 2014], são dois os principais motivos da ineficiência da implementação desta pilha de processamento no *kernel*:

1. **Falta de localidade de conexão:** Grande maioria das aplicações hoje em dia são *multi-threaded* a fim de tirar melhor proveito de sistemas multinúcleos. Entretanto, *sockets* de escuta para conexões são normalmente compartilhados entre os núcleos, criando a necessidade de uma trava para garantir o acesso correto a fila de aceitação de conexões TCP/IP entre os múltiplos núcleos, causando latência no processamento da conexão. Além disso, o núcleo que executa o código de alto nível para processamento TCP/IP dentro do *kernel* não é necessariamente o mesmo que executará o código a nível de Aplicação que recebe e envia dados por meio do *socket*, possivelmente causando mais faltas de cache que também prejudicam a performance.
2. **Espaço de descritores de arquivos compartilhado:** Em sistemas operacionais que seguem a interface POSIX [Group], o espaço dos descritores de arquivos dentro de um processo é compartilhado. No caso do Linux, quando um novo

socket será alocado, é feita uma busca pelo menor valor de descritor de arquivo. Conforme apresentado em [Boyd-Wickizer et al. 2008], em cenários de tráfego intenso nos quais existem muitas conexões concorrentes, essa busca por diferentes *threads* por um descritor de arquivo causa um gargalo significativo devido ao número de faltas de cache e a travas de acesso concorrente a esta estrutura de dados compartilhada entre as *threads*

Além disso, de acordo com [Pesterev et al. 2012], em sistemas *multicores* é comum que múltiplos núcleos do processador interajam no processamento de um pacote e do gerenciamento de uma conexão TCP, causando ineficiências como: invalidações de *cache*, dados sendo copiados entre as *caches* dos núcleos e problemas de performance relacionados a alocação dinâmica de memória, todos estes fatores influenciam negativamente na velocidade de processamento de pacotes. Ainda em máquinas *multicores*, existem estruturas de dados compartilhadas que precisam ser acessadas por diferentes núcleos de processamento, estes acessos devem ser controlados por travas de concorrência e semáforos, que também acabam acarretando em tempo de processamento desperdiçado. Tendo em vista a ineficiência dos atuais *kernels* para o processamento de alta vazão de pacotes, pesquisadores tem estudando duas principais abordagens para acelerar este fluxo: aceleração da *stack* de rede já existente dentro do sistema operacional [Pesterev et al. 2012] e estratégias para desviar o fluxo de processamento de dados do *kernel* para bibliotecas a nível de usuário [Chen and Sun 2018, Jeong et al. 2014, Intel , Rizzo 2012], que será o foco deste estudo. As estratégias do segundo tipo são o que chamamos de Kernel-Bypass, ou seja, um conjunto de técnicas que visa desviar o fluxo de processamento de dados do sistema operacional devido a limitações de performance deste para sistemas de alta vazão. Estratégias de Kernel-Bypass mudam drasticamente arquiteturas tradicionais de Sistemas Operacionais: elas eliminam a atuação do *kernel* no *datapath* de E/S a fim de eliminar sobrecustos operacionais conhecidos no fluxo de processamento de dados pelo *kernel* [Zhang et al. 2021]. De acordo com [Chen and Sun 2018], as técnicas de Kernel-Bypass se dividem em três classes: bibliotecas para processamento de E/S de pacotes em espaço de usuário, *stacks* TCP/IP desenvolvidas em espaço de usuário e *stacks* de rede híbridas. Entretanto, ainda pode-se adicionar mais uma classe a esta classificação; estratégias que combinam componentes de *hardware* e *software* para obter resultados de performance.

2.1. Bibliotecas para processamento de E/S de pacotes a nível de usuário

Estas bibliotecas têm como propósito otimizar o processo de levar um pacote da placa de rede até a aplicação. Por serem feitas a nível de usuário, não possuem muito dos gargalos já comentados anteriormente. Normalmente são empregados mapeamentos de memória da Aplicação para a placa de rede, ou então algum tipo de *driver* específico é disponibilizado para a comunicação com a placa de rede. Por exemplo, a biblioteca DPDK [Intel] funciona criando um mapeamento de memória da Aplicação diretamente para a placa de rede e disponibilizando os PMD, que são *drivers* utilizados para se comunicar com a placa de rede. Assim, removendo o *kernel* por completo da fluxo de processamento. Já a biblioteca netmap [Rizzo 2012] segue uma estratégia parecida com DPDK, entretanto utiliza ainda os *drivers* disponibilizados pelo *kernel* para comunicação com a placa de rede. Importante ressaltar que estas bibliotecas não disponibilizam processamento de protocolos de mais alto nível (e.g., TCP/IP), e sim apenas até a camada L2 do modelo OSI. Os processamentos de mais alto nível são deixados a encargo da Aplicação.

2.2. *Stacks* TCP/IP a nível de usuário

Implementações como mTCP [Jeong et al. 2014] visam reescrever a nível de usuário a *stack* de processamento TCP/IP, a fim de se livrar de sobrecustos operacionais das implementações clássicas disponibilizada pelo *kernel*. Normalmente estas soluções são baseadas em bibliotecas de processamentos de E/S de pacotes em espaço de usuário (discutidas na Seção 2.1) para o processamento de mais baixo nível e implementam a parte mais complexa do processamento TCP/IP. No caso da biblioteca mTCP [Jeong et al. 2014], cada núcleo do processador executa uma *thread* mTCP, gerenciando separadamente estruturas de dados como filas de aceitação de conexão e lista de *sockets*, assim, permitindo mais localidade de conexão [Chen and Sun 2018]. Essa abordagem também permite o uso de estruturas de dados *Lock-free* [Langdale], que eliminam a espera de acesso em estruturas de dados compartilhadas. De acordo com [Chen and Sun 2018], a simples utilização da biblioteca mTCP pode aumentar a performance de aplicações existentes em até 3x.

2.3. *Stacks* de rede híbridas

Um dos problemas das estratégias listadas na Seção 2.2 é que, mesmo que reescrever do zero uma *stack* de rede a nível de usuário proporcione um aumento considerável em performance, normalmente as *stacks* TCP/IP já existentes nos *kernels* atuais são mais atualizadas e menos limitadas do que estas novas implementações [Yasukata et al. 2016]. Por exemplo, a biblioteca mTCP dá suporte apenas ao protocolo TCP e é simples demais para ser colocada em um ambiente de produção realista [Chen and Sun 2018]. Assim, surgiu uma categoria híbrida de implementações Kernel-Bypass que visa utilizar toda a robustez da *stack* de rede disponibilizada pelo *kernel*, ao mesmo tempo que utiliza das bibliotecas de processamento de E/S de pacotes em espaço de usuário (descritas na Seção 2.1) para otimizar a performance de processamento de camadas mais baixas. A

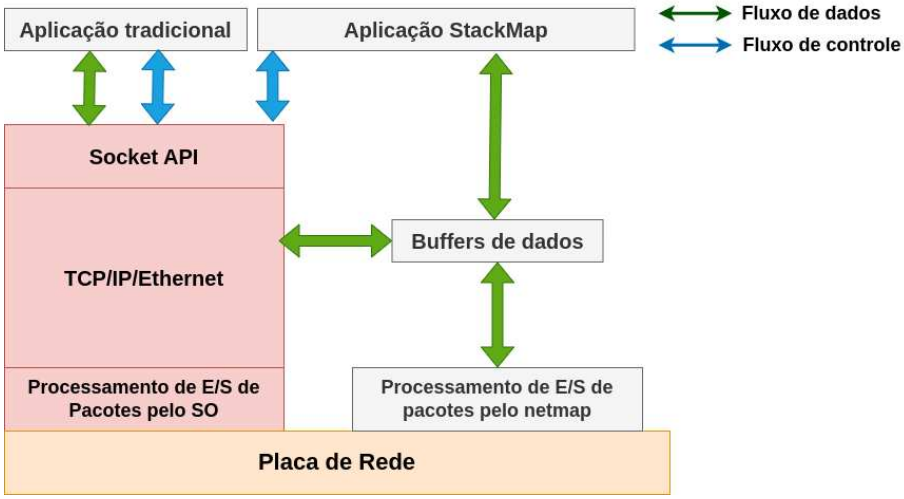


Figure 1. Representação do funcionamento da biblioteca híbrida StackMap

biblioteca StackMap [Yasukata et al. 2016] implementa essa estratégia, utilizando da biblioteca netmap [Rizzo 2012] para o processamento de E/S de pacotes. A Figura 1 exemplifica a estrutura de funcionamento desta biblioteca. Diferente da biblioteca mTCP, ao final do processamento E/S o pacote é enviado para o processamento TCP/IP disponibilizado

pela *kernel*. Dessa forma, StackMap consegue aproveitar da melhoria de performance oferecida pela biblioteca netmap, ao mesmo tempo que pode utilizar dos desenvolvimentos mais modernos dos protocolos TCP/IP implementadas no *kernel*.

2.4. Combinações de *Hardware* e *Software*

Por último, existe a classe de soluções que utilizam de uma combinação não só de bibliotecas a nível de usuário, conforme as soluções anteriores, mas também de dispositivos de *hardware* para realizar otimizações de Kernel-Bypass. Como principal representante desta classe temos os dispositivos de RDMA. Esta tecnologia permite que dois processos em nodos diferentes na rede possam se comunicar diretamente a nível de memória, sem interferência do processador. Diferente das outras estratégias, esta técnica depende de *hardware* específico para este propósito. As placas de rede RDMA seguem a interface `Verbs` [IETF 2003] para realizar a comunicação entre si. Estas placas contam com processamento TCP/IP próprio, não dependendo de implementações destes protocolos pelo *kernel* ou a nível de aplicação.

3. Avaliação Experimental e Resultados

Este capítulo apresenta a comparação de desempenho de duas aplicações Cliente-Servidor; uma utilizando estratégias tradicionais de comunicação de E/S, sendo o fluxo de dados trafegado pelo *kernel* e a outra construída a partir de duas bibliotecas de Kernel-Bypass, fazendo com que o fluxo de dados entre os nodos da aplicação seja trafegado totalmente pelo espaço de usuário, sem envolvimento do *kernel*. Foram criados três cenários para avaliar o desempenho das bibliotecas de Kernel-Bypass em comparação com a pilha de rede do sistema operacional Linux em diferentes situações. Para fins desta deste trabalho, a palavra "requisições" é utilizada para se referir ao envio de uma mensagem para o servidor e sua devida resposta sendo enviada de volta para o cliente, assim, constituindo uma requisição servida. No cenário 1 é explorada a capacidade de escalabilidade no número requisições servidas pelo servidor, aumentando gradativamente o número de servidores disponíveis, dado que a quantidade de conexões clientes está saturada. Este cenário tem como propósito analisar o desempenho de ambas bibliotecas em uma situação de alta concorrência e a escalabilidade no número de requisições servidas, dado o aumento no número de *threads* de servidores. No segundo cenário foi analisada a capacidade de ambas aplicações saturarem a largura de banbda da placa de rede, neste cenário foram estabelecidos casos de teste com duas e quatro *threads* de servidor, variando o tamanho da mensagem enviadas e mantendo um número fixo de conexões clientes. Já no terceiro cenário foi explorado um caso de uso mais comum para aplicações distribuídas, sem necessariamente saturar o número de clientes e levar o servidor ao estresse máximo. Neste cenário manteve-se apenas um servidor e foi-se gradualmente aumentando o número de *threads* clientes, mantendo apenas uma conexão por *thread*, para analisar a relação entre o número de requisições servidas e latência em ambas aplicações.

3.1. Bancada de Testes

Como plataforma de *testbed* foi utilizado o ambiente de testes distribuídos Emulab [White et al. 2002], que disponibiliza infraestrutura sob demanda para execução de experimentos.

Nestes experimentos, foram utilizados nodos denominados d430, compostos por máquinas que possuem dois processadores E5-2630 operando na frequência 2.4GHz, cada um destes processadores equipado 8 núcleos, com 2 *threads* por núcleo, totalizando 16 *threads* por processador. Além disso, possuindo 64GB de memória RAM dividida entre 8 pentes de 8GB e placas de rede com velocidade de transmissão de 10Gb/s. Mais informações sobre este tipo de nodo podem ser encontradas por meio do endereço `https://gitlab.flux.utah.edu/emulab/emulab-devel/-/wikis/Utah-Cluster/d430s`

Em relação à infraestrutura do experimento, utilizou-se duas máquinas conectadas em LAN contando com duas conexões com velocidade de transmissão de 10Gb/s, conforme mostrado na Figura 2. Ambas as máquinas utilizaram o sistema operacional Ubuntu Linux 18.04.

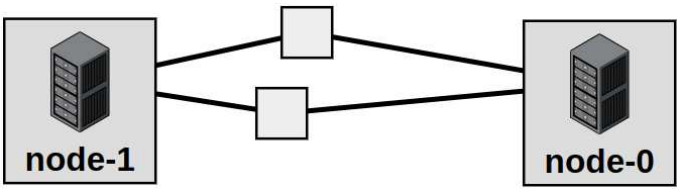


Figure 2. Topologia da rede utilizada para os experimentos.

3.2. Arquitetura da Aplicação de Teste

Para comparar as estratégias de E/S distribuídas, foram implementados duas aplicações tipo Eco. Conforme demonstrado na Figura 3, neste tipo de aplicação o cliente envia uma mensagem para o servidor e este responde a mesma mensagem de volta para o cliente. Para cada uma destas aplicações foram implementados um servidor e um cliente que realizam sua troca de mensagens por meio do protocolo de comunicação TCP. Em um dos protótipos, tanto o cliente quanto o servidor foram implementados a partir das bibliotecas mTCP e DPDK com estratégias de Kernel-Bypass, desviando o fluxo de troca de dados do *kernel* e realizando-o totalmente em espaço de usuário; enquanto isso o outro protótipo utiliza a API de comunicação de rede nativa do *kernel* Linux. As Figuras 4 e 5 ajudam a visualizar como é feita a troca de mensagens entre os protótipos que utilizam de Kernel-Bypass e os que se comunicam de forma tradicional, respectivamente.

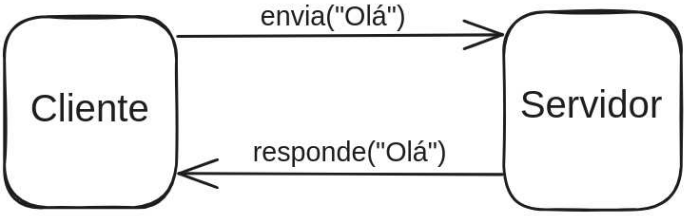


Figure 3. Arquitetura de um sistema Eco.

Ambos protótipos possuem conexões persistentes, ou seja, após estabelecida a conexão entre cliente e servidor, a mesma só é interrompida pelo fim da execução do

processo. Além disso, a troca de dados entre cliente e servidor é contínua, o fluxo de envio e resposta do servidor eco é repetido até o fim do processo.

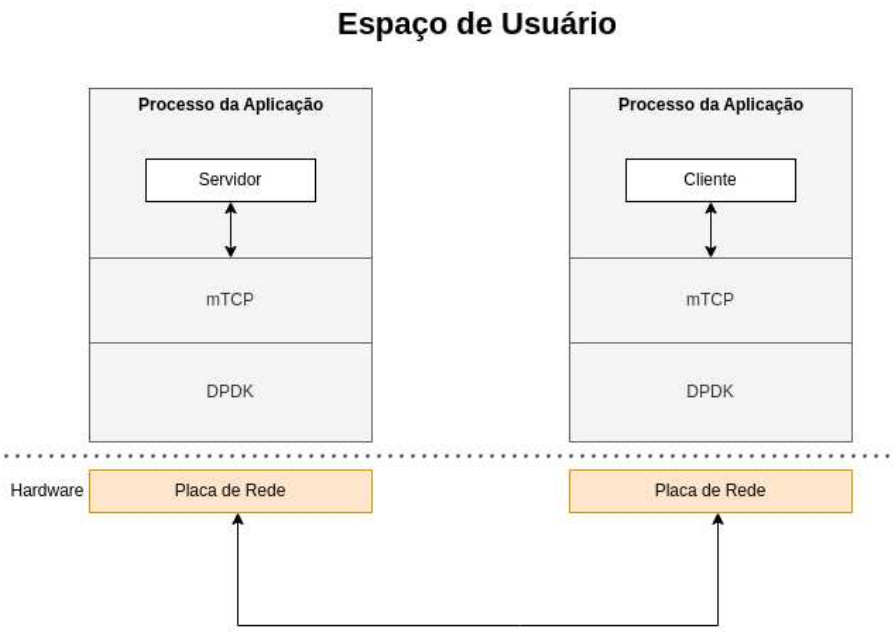


Figure 4. Demonstração do fluxo de troca de dados utilizando as estratégias de Kernel-Bypass.

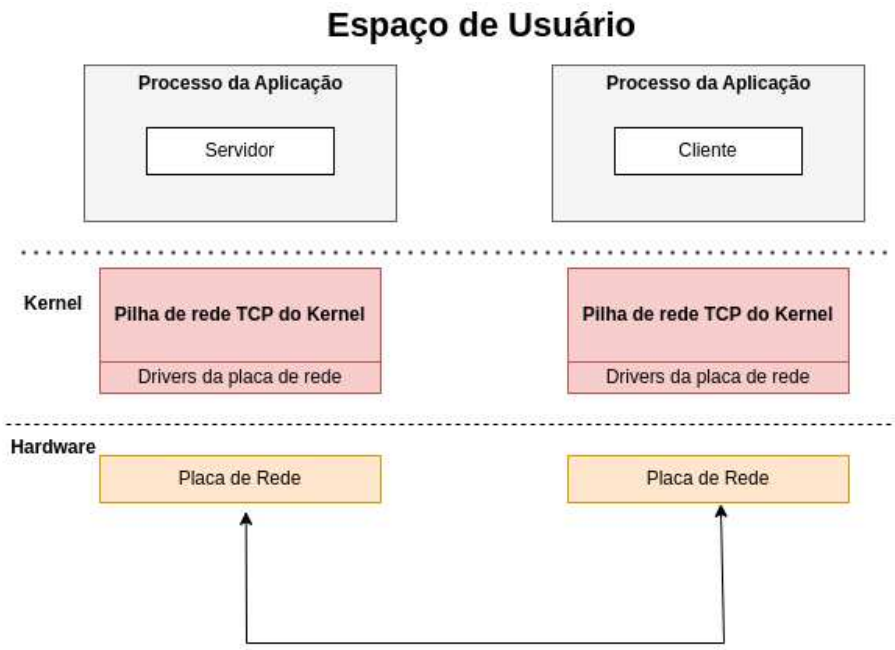


Figure 5. Demonstração do fluxo de troca de dados utilizando a API de rede nativa do Linux.

3.3. Avaliação

Visando explorar o desempenho das estratégias de Kernel-Bypass em diferentes casos de uso foram explorados três cenários: o primeiro simulando um cenário de alta concorrência

e intenso fluxo de dados, no qual é analisada a capacidade da aplicação de lidar com um grande número de conexões simultâneas e como a o número de requisições servidas por esta aplicação escala com a quantidade de servidores disponíveis; o segundo cenário explora a capacidade dos protótipos de saturar a largura de banda da placa de rede com diferentes tamanhos de mensagens enviados e o terceiro cenário simula um cenário de baixa concorrência e baixa demanda dos servidores, analisando o desempenho das aplicações em casos onde o fluxo de troca de dados não é tão intenso.

3.3.1. Cenário 1

Neste cenário foi explorada a capacidade dos protótipos para alta concorrência na troca de mensagens pequenas. O objetivo é analisar a capacidade de escalabilidade de servir requisições TCP dado um cenário saturado de conexões clientes, variando apenas o número de *threads* de servidores. Nestes experimentos, foram utilizadas 8 *threads* clientes, cada uma destas *threads* criando 2000 conexões persistentes com os servidores disponíveis. Após estabelecer a conexão, foi medido o número de requisições servidas pelo servidor em um período de 60 segundos. Este cenário foi divido em dois casos de teste, um enviando mensagens com 64 bytes de tamanho e o outro utilizando mensagens de 128 bytes.

A Figuras 6 e 7 mostram a quantidade total de requisições que foi servida por segundo por cada uma das implementações do protótipo, utilizando tamanhos de mensagens de 64 bytes e 128 bytes. Pode-se perceber que a solução Kernel-Bypass demonstrou grande eficiência no processamento de pacotes de pequeno tamanho, não só conseguindo servir um número maior de requisições com apenas uma *thread* de servidor do que a solução tradicional, mas também possuindo um comportamento linear quando incrementa-se em até 4 o número de *threads* de servidores, a partir de qual o comportamento torna-se menos linear. Enquanto isso, mesmo com a solução tradicional utilizando o *kernel* Linux oferecendo certo grau de escalabilidade, a performance não se compara com a solução Kernel-Bypass, que ofereceu 265% de melhoria no pior caso e 458% no melhor caso.

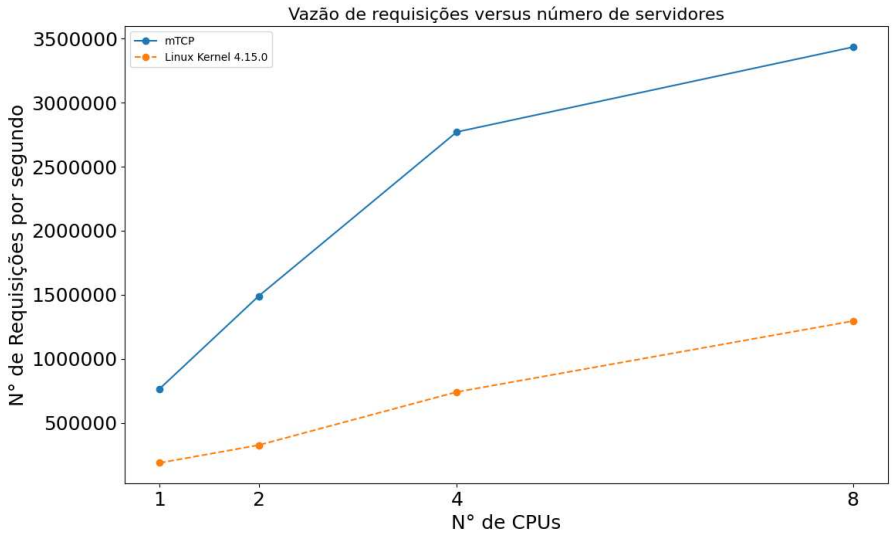


Figure 6. Resultados da execução do Cenário um com tamanho de mensagem de 64 bytes

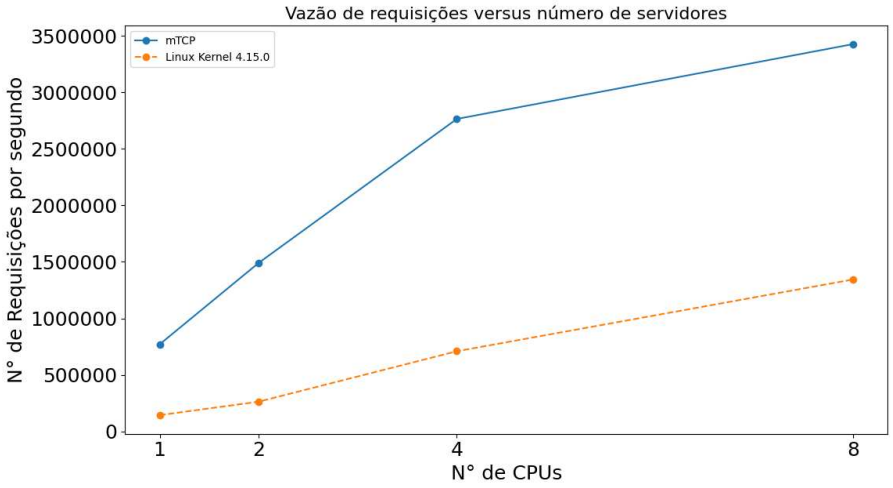


Figure 7. Resultados da execução do Cenário um com tamanho de mensagem de 128 bytes

3.3.2. Cenário 2

Neste cenário foi analisado o potencial de ambos protótipos saturarem a largura de banda da placa de rede. Para isto, foram criados dois casos de teste; um deles utilizando duas *threads* de servidores e o outro utilizando 4 *threads* de servidores, em ambos casos de teste foram utilizados 4 *threads* de clientes, cada cliente contando com 3000 conexões persistentes e mantendo o comportamento da aplicação conforme descrito na Seção 3.2. Após estabelecidas todas as conexões foi monitorada a taxa de transmissão da placa de rede utilizada.

Analisando os resultados apresentados nas Figuras 8 e 9 pode-se perceber que os

casos de teste desenvolvidos a partir das bibliotecas mTCP e DPDK possuem a capacidade de saturar a placa de rede em alguns casos. Enquanto isso, é notável que em nenhum dos casos com a pilha de rede do *kernel* Linux a capacidade máxima da placa de rede foi alcançada, tendo sua vazão máxima de 8.41Gb/s em pacotes de 8192 bytes de tamanho e com duas *threads* de servidores. Percebe-se, na Figura 8, que o caso de testes que utiliza 2 *threads* de servidores do protótipo Kernel-Bypass tem sua performance máxima com mensagens de 1024 bytes, experimentando degradação em vazão com valores de mensagens maiores. Estipula-se que esta queda possa estar associada com as configurações de MTU da placa de rede e a capacidade da biblioteca de lidar com a fragmentação de pacotes. Entretanto, ao analisar o segundo caso de teste, apresentado pela Figura 9, ao dobrar o número de servidores disponíveis, após a vazão máxima ser alcançada com mensagens de 1024 bytes de tamanho, a placa se manteve saturada com tamanhos de pacote maiores, alcançando uma taxa de transmissão de 10Gb/s. Aprofundando a análise da Figura 9, vemos que, diferente da implementação Kernel-Bypass, ao dobrar o número de servidores disponíveis na solução tradicional, não temos um aumento na vazão máxima do protótipo, e sim o efeito contrário, baixando a vazão máxima para 5.72Gb/s em casos de mensagens com tamanho de 4096 bytes, representando uma queda de quase 30% do caso de teste com 2 servidores.

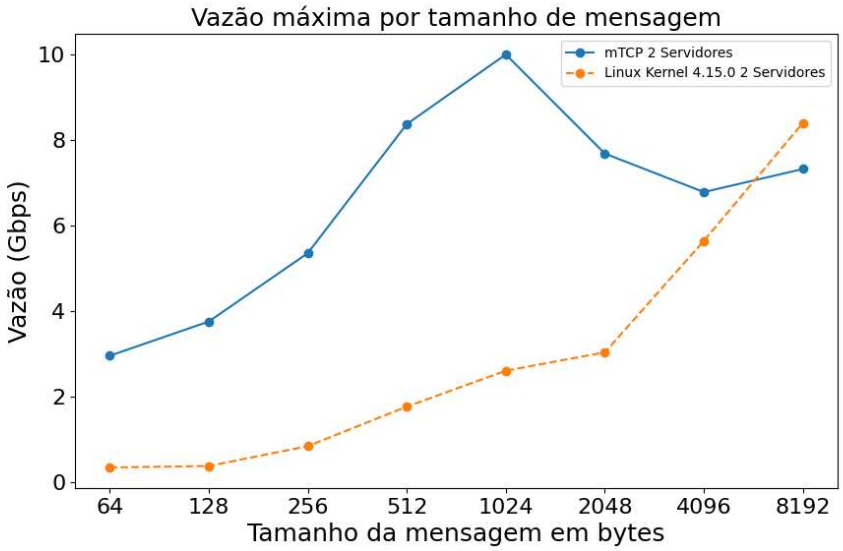


Figure 8. Execução do segundo cenário utilizando 2 servidores

3.3.3. Cenário 3

Neste cenário optou-se por explorar um caso mais comum de desenvolvimento de aplicações distribuídas, sem necessariamente saturar a quantidade de conexões de clientes. Optou-se por fixar apenas um servidor e gradativamente aumentar o número de *threads* clientes, entretanto, ao invés de cada *thread* cliente possuir diversas conexões como nos exemplos anteriores, cada *thread* cliente contou com apenas uma conexão, cada uma destas conexões enviando mensagens de 2048 bytes. Para este cenário, foram utilizadas respectivamente 1, 2, 4, 8 e 16 *threads* clientes, coincidindo com os pontos no

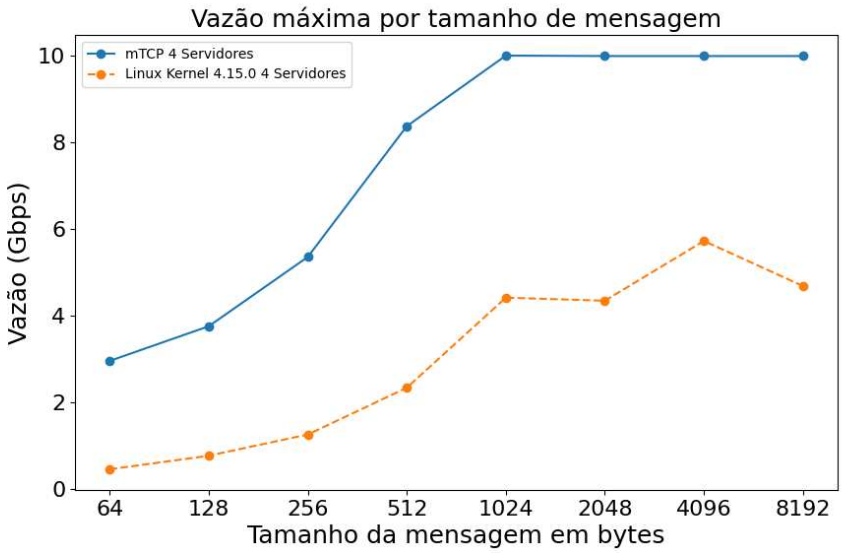


Figure 9. Execução do segundo cenário utilizando 4 servidores

eixo x da Figura 10 de cada um dos protótipos desenvolvido. No caso do protótipo de Kernel-Bypass, a cada nova *thread* cliente foi também criada uma nova *thread* mTCP. Após encerrar o experimento, foi utilizado o percentil 95 das latências medidas pelos protótipos para a análise de resultados. Analisando a Figura 10 pode-se perceber que a biblioteca mTCP teve um desempenho inferior quando comparada a pilha de rede Linux, acarretando um aumento de latência de diminuição na vazão de requisições servidas. Enquanto isso, o protótipo que utiliza estratégias tradicionais de E/S teve impacto mínimo na sua latência conforme a vazão aumentava devido ao aumento no número de *threads* clientes, a aplicação desenvolvida a partir das bibliotecas mTCP e DPDK demonstrou degradação significativa na latência, alcançando um aumento de 18x no pior caso.

Estipula-se que esta queda em performance deve-se ao custo operacional acarretado para manter uma *thread* mTCP; conforme a implementação da biblioteca, cada uma destas *threads* é responsável por realizar todo o fluxo de processamento TCP/IP, gerenciando trocas de mensagens entre a *thread* de aplicação e a placa de rede. Conforme viu-se no Cenário 1, a biblioteca mTCP possui grande potencial quando criamos um grande número de conexões por *thread* mTCP, entretanto, ao criar apenas uma conexão nesta *thread* estamos lidando com todo o custo operacional causado pela mesma sem poder tirar total proveito de suas vantagens em casos de alta concorrência.

4. Conclusão

Neste trabalho buscou-se estudar e apresentar os gargalos conhecidos no processamento de rede oferecido pelo *kernel* Linux ao mesmo tempo em que buscava-se uma alternativa mais performática. Ao aprofundar os estudos nas estratégias de Kernel-Bypass, notou-se que muito dos gargalos conhecidos poderiam ser resolvidos por meio da implementação de funcionalidades tradicionalmente do *kernel*, como por exemplo processamento TCP/IP, em espaço de usuário na forma de bibliotecas.

A fim de validar o desempenho das estratégias de Kernel-Bypass foram escolhi-

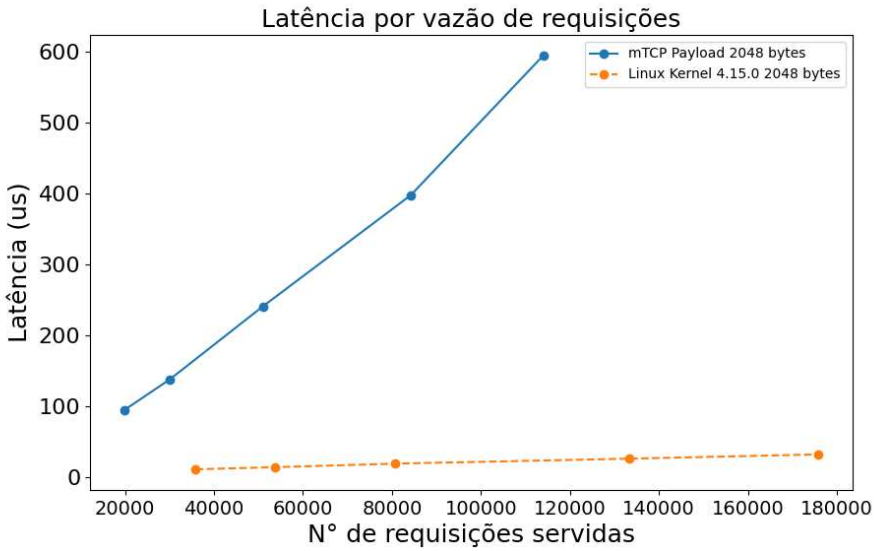


Figure 10. Gráfico de latência versus vazão de requisições para mensagens com 2048 bytes de tamanho.

das as bibliotecas DPDK e mTCP, responsáveis respectivamente por processamento de pacotes e processamento TCP/IP, para desenvolver um protótipo que trafega dados entre aplicações distribuídas completamente pelo espaço de usuário, sem interferência do *kernel* na troca de informação. A partir da implementação deste protótipo uma série de testes em diferentes cenários foi realizada para analisar métricas como vazão e latência e compará-las a um protótipo desenvolvido utilizando a API de rede tradicional do Linux, ou seja, trafegando os dados pelo *kernel*.

Por meio dos cenários de teste provou-se que as bibliotecas DPDK e mTCP possuem escalabilidade e desempenho em vazão superiores a pilha de rede do *kernel* Linux quando realiza-se processamento de pacotes pequenos e com um grande número de conexões persistentes. Além disso, o protótipo Kernel-Bypass mostrou-se capaz de saturar a largura de banda da placa de rede, enquanto a aplicação desenvolvida utilizando *kernel* Linux não foi capaz do mesmo feito. Entretanto, em cenários de pouca concorrência a aplicação Kernel-Bypass foi inferior na relação latência x vazão quando comparada a outra implementação, esta degradação em performance apenas salienta que este tipo de estratégia tem como foco ser aplicada em aplicações que antecipam alta vazão de dados e um grau alto de concorrência, não sendo adequadas para aplicações que não tem como objetivo um intenso fluxo de dados.

5. Trabalhos Futuros

Mesmo este trabalho tendo explorado de forma teórica algumas estratégias de Kernel-Bypass a avaliação experimental foi limitada a apenas um conjunto de duas bibliotecas específicas. Esta limitação abre portas para a exploração de mais implementações de bibliotecas e dispositivos Kernel-Bypass disponíveis. Um exemplo seria aprofundar o conhecimento em dispositivos RDMA, que permitem acesso à memória de um computador para a memória de outro computador sem envolver o sistema operacional de nenhum dos computadores, este tipo de técnica é dependente de placas de rede específicas. Ou então

a biblioteca F-Stack [Jiang], que também oferece uma implementação dos protocolos TCP/IP similar a mTCP e totalmente a nível de usuário, sem interferência do *kernel*.

Além disso, pensando nas bibliotecas mTCP e DPDK, poderiam ser feitas migrações de sistemas existentes que atualmente utilizam como camada de rede o *kernel* Linux para utilizarem estas bibliotecas, a fim de medir o impacto do uso dessas estratégias em um sistema real. Um exemplo seria a biblioteca `librdkafka` [Edenhill], que implementa o protocolo Apache Kafka de Produtores/Consumidores na linguagem de programação C. Ainda tendo as bibliotecas mTCP e DPDK em mente, seria possível explorar placas de rede com uma largura de banda maior para obter uma noção exata do limite da taxa de transferência de dados do protótipo teste. Conforme observado nos testes, a aplicação exemplo desenvolvida foi capaz de alcançar a taxa de vazão de 10Gbp/s, sendo o único limitante o próprio limite de largura de banda da placa.

Outra abordagem interessante, mantendo-se coerente a proposta de estratégias para otimização de operações de E/S em sistemas distribuídos de alta vazão, poderia ser a investigação de algoritmos com tratamento eficiente para operações de E/S, como por exemplo estudo de algoritmos *cache-oblivious* que buscam tirar melhor proveito da memória cache sem ter como pré-requisito a configuração da mesma. Ou ainda explorar o uso de estruturas de dados eficientes para operações de E/S, como a árvores de *buffers* [Arge 2003], um tipo especial de árvore binária que possui operações otimizadas para E/S.

Por fim, tendo em vista as ineficiências no processamento de rede do *kernel* Linux apresentadas neste trabalho, outra abordagem válida seria estudar mais a fundo o *kernel* Linux e propor possíveis alterações no projeto do atual deste a fim de sanar os gargalos encontrados.

References

Arge, L. (2003). The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24.

Barroso, L., Marty, M., Patterson, D., and Ranganathan, P. (2017). Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54.

Boyd-Wickizer, S., Chen, H., Chen, R., Mao, Y., Kaashoek, F., Morris, R., Pesterev, A., Stein, L., Wu, M., Dai, Y., Zhang, Y., and Zhang, Z. (2008). Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 43–57, USA. USENIX Association.

Chen, R. and Sun, G. (2018). A survey of kernel-bypass techniques in network stack. In *Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence*, CSAI '18, page 474–477, New York, NY, USA. Association for Computing Machinery.

Cox, A. (1996). Network buffers and memory management.

Edenhill, M. librdkafka - the apache kafka c/c++ client library.

Group, T. O. The single unix specification version 3 - overview.

IETF (2003). Rdma protocol verbs specification.

Intel. Data plane development kit.

Jeong, E. Y., Woo, S., Jamshed, M., Jeong, H., Ihm, S., Han, D., and Park, K. (2014). Mtcp: A highly scalable user-level tcp stack for multicore systems. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, page 489–502, USA. USENIX Association.

Jiang, J. F-stack.

Langdale, G. Lock-free programming.

Pesterev, A., Strauss, J., Zeldovich, N., and Morris, R. T. (2012). Improving network connection locality on multicore systems. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, page 337–350, New York, NY, USA. Association for Computing Machinery.

Rizzo, L. (2012). Netmap: A novel framework for fast packet i/o. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, page 9, USA. USENIX Association.

Tsuna (2010). How long does it take to make a context swtich? [Online; acessado em 24/05/2023].

White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A. (2002). An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA. USENIX Association.

Yasukata, K., Honda, M., Santry, D., and Eggert, L. (2016). StackMap: Low-Latency networking with the OS stack and dedicated NICs. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 43–56, Denver, CO. USENIX Association.

Zhang, I., Raybuck, A., Patel, P., Olynyk, K., Nelson, J., Leija, O. S. N., Martinez, A., Liu, J., Simpson, A. K., Jayakar, S., Penna, P. H., Demoulin, M., Choudhury, P., and Badam, A. (2021). The demikernel datapath os architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 195–211, New York, NY, USA. Association for Computing Machinery.

APÊNDICE B – CÓDIGO FONTE

O código fonte desenvolvido neste projeto, em sua integralidade, está disponível no respositório do autor no GitHub, no seguinte link: `https://github.com/Djsouls/tcc.git`. Lá está descrito como executar tanto os experimentos como a aplicação.