# eQUIC Gateway: Maximizando a vazão de pacotes do protocolo QUIC através de um serviço de gateway utilizando eBPF + XDP

Gustavo Pantuza
*Depart. de Ciência da Computação*
*Universidade Federal de Minas Gerais*
Belo Horizonte, Brasil
pantuza@dcc.ufmg.br

Marcos A. M. Vieira
*Depart. de Ciência da Computação*
*Universidade Federal de Minas Gerais*
Belo Horizonte, Brasil
mmvieira@dcc.ufmg.br

Luiz F. M. Vieira
*Depart. de Ciência da Computação*
*Universidade Federal de Minas Gerais*
Belo Horizonte, Brasil
lfvieira@dcc.ufmg.br

***Resumo*—O protocolo QUIC é considerado um ambiente de experimentação e uma evolução do protocolo TCP. Aplicações criadas através do QUIC em substituição a tradicional pilha HTTPS tem demonstrado ganhos em desempenho. A técnica de transferência (offload) de carga computacional para o espaço de núcleo é utilizada como otimização em aplicações modernas e traz consigo desafios arquiteturais e algorítmicos. Esse trabalho apresenta o eQUIC Gateway, um módulo de bloqueio de pacotes em espaço de núcleo que utiliza informações fornecidas por uma aplicação QUIC em espaço de usuário em tempo real. Através da transferência (*offload*) da carga computacional de bloqueio de pacotes para o espaço de núcleo, o eQUIC Gateway aumentou a vazão de pacotes em 30,9%, reduziu em 65% a duração média em requisições HTTPS sob ataque e reduziu em 26,4% o tempo de CPU para bloquear-se pacotes.**

***Index Terms*—Sistemas distribuídos, Protocolos de comunicação, Otimização, Avaliação de desempenho, Redes de computadores**

## I. Introdução

O protocolo QUIC [1] é utilizado e experimentado desde 2012 e motiva-se em ser um substituto ao HTTPS (RFC-2818). Conforme discutido por Adam em [2], o QUIC foi projetado em espaço de usuário pois permite ser facilmente utilizado e modificado em diversos sistemas operacionais.

Sendo implementado sobre o protocolo UDP, isso consegue evitar que equipamentos de rede ao longo da Internet alterem ou interfiram no tráfego QUIC. Atualmente em rascunho, a especificação formal do protocolo QUIC está disponível em [1]. Assim como o protocolo TCP, o QUIC é um protocolo orientado a conexão, confiável e com fluxos de pacotes ordenados. No entanto, diferentemente do TCP, o QUIC permite multiplexação dos fluxos de pacotes em uma mesma conexão. Além disso é otimizado para ser eficiente no controle da latência [3].

O QUIC utiliza o protocolo TLS (RFC-5246) para construir um protocolo de transporte criptografado. Contudo, quando comparado ao HTTPS, reduz a zero o número de transmissões e confirmações (*Round Trip*) necessárias para o acordo da conexão QUIC (*handshake*). Apesar da vantagem no estabelecimento de conexões, o protocolo QUIC gasta 50% do tempo de CPU em computação criptográfica [4]. A utilização de *offload* (transferência da carga computacional) para interfaces de redes inteligentes mostrou ganhos de pelo menos 2.3x no desempenho do protocolo TCP [4].

O problema endereçado nesse trabalho é melhorar o desempenho do protocolo QUIC através da transferência de computação para o espaço de núcleo (*kernel*) do sistema operacional. Este problema apresenta alguns desafios: as mensagens do protocolo QUIC são criptografadas, o que dificulta saber o estado da conexão. Além disso, a programação em espaço de núcleo é complexa e de difícil depuração.

Empresas de tecnologia de Internet que lidam com grandes volumes de clientes simultâneos como GoDaddy, Google, Facebook e Microsoft lidam com o impacto da latência de cauda [5]. Esse tipo de problema, em escala, motiva o presente trabalho a criar soluções de bloqueio de pacotes que tentam reduzir a carga de trabalho sem impactar a latência de clientes que não devem ser bloqueados.

As contribuições deste trabalho são: a apresentação do eQUIC *Gateway*, um módulo de bloqueio de pacotes em espaço de núcleo que utiliza informações fornecidas pela aplicação QUIC em espaço de usuário e em tempo real e a biblioteca eQUIC para carregamento dinâmico de programas eBPF. Através da transferência (*offload*) da carga computacional para o núcleo, mostra-se um **aumento na vazão** de pacotes, **redução na duração média** em requisições HTTPS, **redução do tempo de CPU** necessário para decidir-se um bloqueio e **redução no número de chamadas de sistemas** executadas pela aplicação.

Este artigo está dividido conforme descrito à seguir. Primeiro descreve-se o funcionamento do protocolo QUIC. Em seguida são explicadas a combinação das ferramentas XDP e eBPF, tecnologias que permitem a transferência de computação do espaço de usuário para o espaço de núcleo. Após esta introdução às ferramentas desse trabalho, comparamos o estado da arte com a presente proposta. Em seguida são apresentadas a arquitetura da solução proposta e o funcionamento do eQUIC *Gateway*. Depois, são descritos os experimentos e a análise dos resultados obtidos. Ao final

discutem-se os trabalhos futuros e a conclusão.

## II. O protocolo QUIC

Proposto como um novo protocolo de transporte, o QUIC foi desenhado para melhorar o desempenho de tráfego HTTPS e permitir a evolução contínua e de maneira rápida de novos mecanismos experimentais de transporte.

Na proposta apresentada por [2], o QUIC substitui os protocolos utilizados em aplicações HTTPS: HTTP/2 (RFC-7540), TLS (RFC-5246) e TCP. O QUIC é construído sobre o protocolo de transporte UDP. Criptografado por padrão, dificulta a modificação dos pacotes por equipamentos de rede ao longo da Internet.

O trabalho feito por Arach [6] discute a trajetória do protocolo QUIC enquanto evolução experimental do protocolo TCP. Uma avaliação do protocolo entre 2016 e 2018 por um ISP na Europa, mostrou que o número de endereços IPv4 com QUIC habilitados triplicou no período [3] e que a empresa Google, individualmente, publicava aproximadamente 42,1% do seu tráfego via QUIC.

O trabalho feito por Palmer [7] apresenta como, através do protocolo QUIC, foi possível melhorar o desempenho em uma aplicação de transmissão contínua (*streaming*) de vídeo. Em outro trabalho, apresentado por Sivakumar [8], o protocolo QUIC é utilizado para fazer controle de congestionamento utilizando inteligência artificial.

Esses trabalhos supracitados mostram como o protocolo QUIC pode ser utilizado por diversos tipos de aplicações para fins distintos.

## III. Fundamentação teórica

O eBPF (*Extended Berkeley Packet Filter*) [9] é um conjunto de instruções e uma máquina virtual para programas no espaço de núcleo do sistema operacional Linux [10]. Ele permite modificação, interação, metrificação e análise em tempo de execução de pacotes de rede de forma eficiente, isolada e minimizando a troca de contexto. Isto permite que o eBPF seja uma ferramenta para o processamento rápido de pacotes de rede (*Fast Packet Processing*) [11].

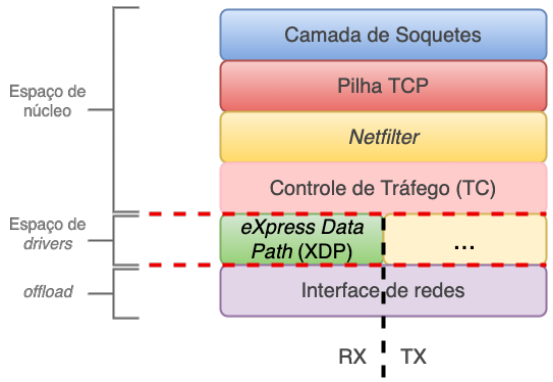


Figura 1. Pilha de implementações de redes no núcleo do sistema operacional Linux. O pontilhado em vermelho demonstra a camada à qual o programa eBPF será carregado

O XDP (eXpress Data Path) é um recurso adicionado ao núcleo do Linux de modo a permitir que programas possam processar pacotes de rede no mesmo contexto e camada dos programas controladores *drivers* de dispositivos [12]. Este recurso é muitas vezes citado como gancho XDP (*XDP Hook*).

A criação do gancho XDP possibilitou otimizar diversas aplicações através de transferência *offload* de computações para o espaço de núcleo utilizando a máquina virtual BPF [13].

A implementação da pilha de protocolos Internet do sistema operacional Linux permite anexar programas eBPF em várias camadas dessa pilha. A imagem 1 apresenta em qual camada o XDP se encontra.

## IV. Trabalhos relacionados

O projeto AccelTCP [4] apresenta uma implementação de pilha TCP assistida por interfaces de redes inteligentes (*smart nics*) cuja arquitetura transfere parte da complexidade do protocolo TCP para ser processado diretamente na interface de rede utilizando um programa P4 [14].

Este trabalho também transfere parte da complexidade para outro ambiente que não o espaço de usuário. No entanto, transfere-se responsabilidades para o núcleo do sistema operacional Linux e não para a interface de redes inteligente. Além disso, trabalha-se com o protocolo QUIC e não o TCP.

O artigo apresentado por Yang [15] discute em profundidade os desafios para se fazer transferência (*offload*) do protocolo QUIC para uma interface de redes inteligente. São apresentadas limitações das atuais interfaces disponíveis no mercado e também das diferentes implementações do protocolo. O trabalho propõe uma arquitetura à qual uma interface de redes poderia utilizar para fazer a transferência (*offload*) do protocolo QUIC de forma eficiente. Esse trabalho colabora com o presente artigo apresentando limitações importantes como a intensa computação do módulo criptográfico do protocolo QUIC e o custo de 50% do tempo de CPU dedicado apenas a troca de contexto entre o núcleo do sistema operacional e o programa em espaço de usuário.

Outro artigo feito por Wang [16] implementa o protocolo QUIC no kernel e compara ao TCP focando no controle de congestionamento de pacotes. Contudo, o trabalho não implementa e nem avalia a parte criptográfica necessária pelo protocolo QUIC e que, como citado anteriormente, demanda 50% do tempo do processamento. O presente trabalho segue uma abordagem híbrida entre espaço de *kernel* e de usuário.

O trabalho feito por Coninck [17] apresenta o PQUIC (*Pluginizing QUIC*), uma ferramenta para estender o protocolo QUIC de modo a permitir que pesquisadores/programadores criem *plugins* na camada das conexões do protocolo. Este projeto difere-se do presente trabalho, pois utiliza uma versão da máquina virtual BPF no espaço de usuário chamada uBPF [18]. O presente trabalho utiliza a máquina virtual no espaço de núcleo para reduzir troca de contexto no sistema operacional Linux para melhorar o desempenho.

Uma argumentação apresentada no trabalho Arrakis [19] descreve o sistema operacional como um plano de controle

para aplicações em rede. Nessa abordagem, o Linux foi modificado de modo a dar acesso direto a dispositivos virtualizados desviando completamente (*bypass*) do núcleo do sistema operacional. Esse desvio é base fundamental para o presente trabalho, que vai no caminho inverso: reduzindo a computação no espaço de usuário e delegando esta responsabilidade ao núcleo.

Esse trabalho segue o caminho inverso ao utilizado em soluções como DPDK [20], que desviam do núcleo do sistema operacional (*kernel bypass*) permitindo que a aplicação possa comunicar-se diretamente com a interface de redes. O *kernel bypass* elimina toda a sobrecarga de trabalho (*overhead*) injetado pelo *kernel* do Linux ao processar pacotes. Trabalhos como o Shenango [21] e Caladan [22] monstram como é possível reduzir latência e CPU ao processar pacotes com DPDK. No entanto, para problemas como o do presente trabalho, o DPDK exigiria que toda a pilha TCP/IP que precede (nas camadas abaixo) o QUIC fosse reescrita. É nesse ponto que eBPF + XDP fazem sentido, pois permitem que o caminho inverso (*kernel offload*) alcance resultados semelhantes ao *kernel bypass* sem grandes modificações nos programas. Trabalhos como [23] corroboram com a abordagem do presente trabalho e demonstram como o *offload* de programas para processamento de pacotes através de eBPF podem operar em taxa de linha em uma solução *serverless*, tecnologia descrita em detalhes por Vieira em [24].

## V. Arquitetura Geral

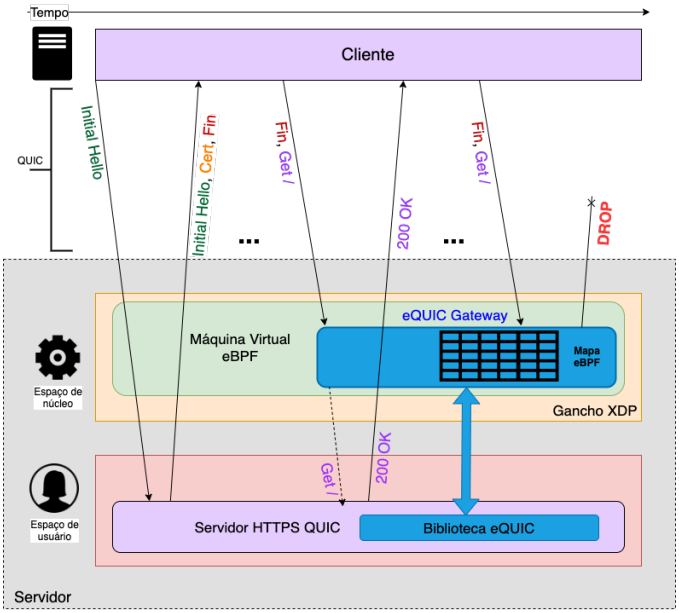


Figura 2. Transferência da computação de bloqueio de fluxos para o espaço de núcleo através XDP + eBPF e da biblioteca eQUIC utilizando mapas eBPF para comunicação entre espaço de usuário e de núcleo.

Um serviço que atua como entrada (*Gateway*) em um sistema distribuído tem como característica fazer verificações nos pacotes endereçados à aplicação de destino de modo a controlar quais pacotes estão autorizados a prosseguir.

No contexto do presente trabalho, tem-se dois programas: um programa eBPF a ser carregado no gancho XDP dentro da máquina virtual BPF do núcleo do sistema operacional Linux e uma biblioteca em espaço de usuário que permite aplicações se comunicarem com o módulo no núcleo através de mapas eBPF.

Ao receber um pacote QUIC, o programa eBPF do núcleo analisa os fluxos verificando mapas eBPF preenchidos pelo programa em espaço de usuário. Quando o fluxo está em conformidade com as regras da aplicação, os pacotes deste fluxo são autorizados a prosseguir na pilha de protocolos do sistema operacional até chegar à aplicação em espaço de usuário. Caso contrário, os pacotes do fluxo são bloqueados diretamente no núcleo do sistema operacional pelo programa eBPF anexado ao gancho XDP. Ou seja, pelo eQUIC *Gateway*.

A figura 2 exemplifica a comunicação QUIC dentro da arquitetura em sistema operacional Linux atravessando o programa eBPF + XDP no núcleo e depois chegando ao programa em espaço de usuário. Tome como exemplo o controle de quotas de conexões de um único cliente do servidor QUIC. Ao atingir a quota o eQUIC *Gateway* passa a descartar os pacotes desse cliente.

## VI. QUIC Gateway

O serviço de *Gateway* é dividido em duas partes:

- 1. Programa eBPF carregado no gancho XDP;
- 2. Biblioteca em espaço de usuário para comunicação com o núcleo;

As seções subsequentes explicam cada parte desse serviço.

### A. O programa eBPF

Programas eBPF podem criar mapas em memória principal e compartilhá-los com programas em espaço de usuário. O programa eBPF deste trabalho faz uso destes mapas para permitir que o programa em espaço de usuário possa controlar o tráfego entrante. Tomando como exemplo o controle de conexões simultâneas por um mesmo cliente, o programa eBPF bloqueia fluxos de um cliente do servidor baseado nos mapas eBPF modificados e preenchidos pela aplicação em espaço de usuário. Estes mapas são tabelas *Hash* carregadas estaticamente na carga do programa eBPF.

Considere que o programa em espaço de usuário atua como plano de controle e o programa no núcleo como plano de dados programável:

**Algorithm 1** Pseudo-Código de bloqueio de pacotes por quota de conexões simultâneas

```
0: procedure QUICQuota(packet)
0:     ethernet ← GetEthernet(packet)
0:     ip ← GetIP(ethernet)
0:     udp ← GetUDP(ip)
0:     nConns ← LookupQuotasMap(ip)
0:     if nConns > QUOTA_LIMIT then
0:         return XDP_DROP
0:     return XDP_PASS
=0
```

O pseudo-código acima exemplifica o caminho feito pelo programa em espaço de núcleo para bloquear pacotes de um mesmo cliente uma vez que ele tenha atingido o limite de conexões simultâneas permitido. Como o gancho XDP é executado pela máquina virtual BPF assim que o controlador (*driver*) da interface de redes é sinalizada com a chegada de um novo pacote, o programa acima tem a possibilidade de bloquear o pacote antecipadamente.

A figura 2 exemplifica o caminho feito pelos pacotes que chegam ao servidor. Note que o programa de núcleo apenas verifica se a quota de conexões simultâneas foi alcançada. Caso positivo, passa a descartar os pacotes daquele fluxo. Caso contrário, os pacotes podem seguir até a aplicação, que no caso de uma aplicação HTTPS, retorna '200 OK' conforme exemplificado na figura 2.

### B. A biblioteca eQUIC

A biblioteca eQUIC é uma camada para programas em espaço de usuário que necessitem carregar programas no núcleo do sistema operacional de forma transparente utilizando eBPF + XDP. Além de cuidar da carga e descarga do programa em espaço de núcleo, a biblioteca fornece funções para atualização dos mapas eBPF de forma concorrente. Isso permite a comunicação assíncrona entre os dois contextos do sistema operacional.

Deste modo, tomando o exemplo do controle de quota de conexões, um programa em espaço de usuário pode utilizar a biblioteca eQUIC para atualizar mapas eBPF instruindo o programa do núcleo quando ele deve bloquear pacotes de um determinado fluxo. A biblioteca eQUIC garante que o ao interromper a execução do programa em espaço de usuário, o código eBPF seja descarregado do gancho XDP e desassociado da interface de redes à qual o programa fora anexado.

De forma plugável a biblioteca eQUIC pode ser chamada em espaço de usuário, por exemplo, como no código em linguagem C abaixo:

```c
int main (int argc, char **argv)
{  /* Some User Space application code */

  /* eQUIC initialization and setup */
   equic_t equic;
   equic_get_interface(&equic, "eth0");
   equic_read_object(&equic, "equic_kern.o");

   signal(SIGINT, equic_sigint_callback);
   signal(SIGTERM, equic_sigterm_callback);

   equic_load(&equic);

   /* More User Space application code */
}
```

### C. Interação entre programas

Como descrito nessa seção, o *eQUIC Gateway* tem duas partes: uma no núcleo do sistema operacional e outra no espaço de usuário. Para que um programa se comunique com o outro utiliza-se mapas eBPF, um recurso de compartilhamento de estruturas de dados em memória principal à qual ambos programas podem ler e escrever. A seta azul na figura 2 ilustra essa comunicação.

Na implementação do *eQUIC*, apenas o espaço de usuário escreve no mapa que controla as quotas de conexões simultâneas de um mesmo cliente. Duas funções da biblioteca eQUIC de espaço de usuário permitem que um programa execute essas operações no mapa: $equic\_inc\_counter()$ e $equic\_dec\_counter()$.

O programa em espaço de núcleo (eBPF + XDP), por motivos de desempenho, apenas lê os contadores de conexões simultâneas do cliente cujo o pacote acabou de chegar ao gancho XDP. O algoritmo 1 mostra como o bloqueio é feito no núcleo.

## VII. Experimentos

Como objeto de experimentação e avaliação deste trabalho utilizou-se uma aplicação HTTPS que retorna texto no corpo de suas requisições. Esta aplicação foi desenvolvida utilizando o protocolo QUIC através da biblioteca *lsquic* [25].

Os experimentos são baseados em duas versões desta aplicação: uma com o controle de quotas inteiramente feito em espaço de usuário e outra com o uso do *Gateway* proposto por este trabalho.

Foram executados 3 experimentos comparativos conforme descritos à seguir:

- **Requisições por segundo**: Computar a vazão média de requisições que a aplicação suporta sem nenhum bloqueio de conexões simultâneas, com bloqueio em espaço de usuário e com bloqueio em espaço de núcleo;
- **Duração das requisições**: Computar em média quanto tempo em milissegundos as requisições atendidas pelo servidor duraram entre a inicialização e terminação da conexão com bloqueio em espaço de usuário e espaço de núcleo;
- **Tempo de CPU**: Computar quanto tempo em microssegundos cada programa leva para efetuar o bloqueio de um cliente que tenha ultrapassado a quota de conexões simultâneas;

No contexto deste experimento, cada conexão contém apenas uma requisição HTTPS para qual o servidor sempre irá retornar um corpo de 256 *kilobytes* de dados.

O experimento é composto por um servidor QUIC implementando uma aplicação HTTPS e oito clientes. Cada cliente envia mil requisições ao servidor. Quatro destes clientes fazem suas requisições sequencialmente. Os quatro clientes restantes enviam requisições paralelamente.

Os clientes são sempre executados simultaneamente. No entanto, os quatro clientes que executam em paralelo aguardam 30 segundos antes de começar a enviar suas requisições (ataque). Estes quatro irão sofrer controle de quota. Estes experimentos são baseados em contêineres através do utilitário [26]. Tanto o servidor quanto os clientes são executados em uma rede com nove contêineres: um servidor e oito clientes. Todos em uma rede local.

## VIII. Análise

Essa seção apresenta e discute os resultados das análises feitas em cima dos três experimentos propostos.

### A. Requisições por segundo

A figura 3 mostra a vazão (*throughput*) de requisições HTTPS durante a execução do experimento para ambas implementações: em espaço de usuário e espaço de núcleo.

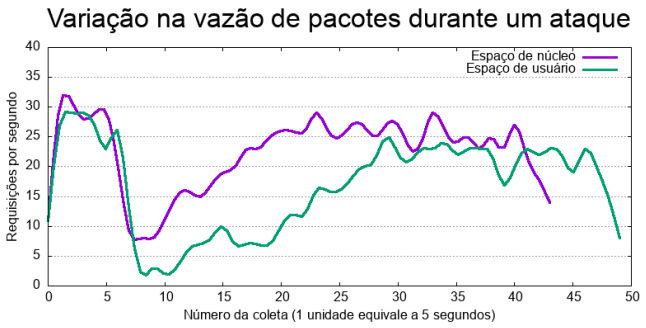


Figura 3. Evolução da vazão de requisições durante toda a execução do experimento para bloqueio feito em espaço de usuário e bloqueio feito em espaço de núcleo.

O gráfico na figura 3 apresenta uma degradação da vazão ao alcançar o valor 5 nas abscissas. Este valor equivale ao momento em que os quatro clientes que executam em paralelo começam a enviar suas requisições. Note, que a degradação na implementação em espaço de usuário é mais acentuada do que na implementação em espaço de núcleo.

As curvas mostram que a implementação em espaço de núcleo recuperou sua vazão no valor 22 das abscissas. Já a implementação em espaço de usuário somente recuperou-se no valor 27 das abscissas. Como cada unidade das abscissas equivale a 5 segundos decorridos no experimento, pode-se afirmar que, para este experimento, a implementação em espaço de núcleo se recuperou do ataque 18% mais rápido do que a implementação em espaço de usuário.

É possível notar também que o tempo total para computar todos as requisições foi menor na implementação em espaço de núcleo, uma vez que sua vazão foi, em média, maior.

A figura 4 mostra a média harmônica calculada para todo o experimento acima. É possível notar um **aumento na vazão** de pacotes em **30,9%** quando se utiliza o bloqueio feito no espaço de núcleo pelo QUIC *Gateway*.

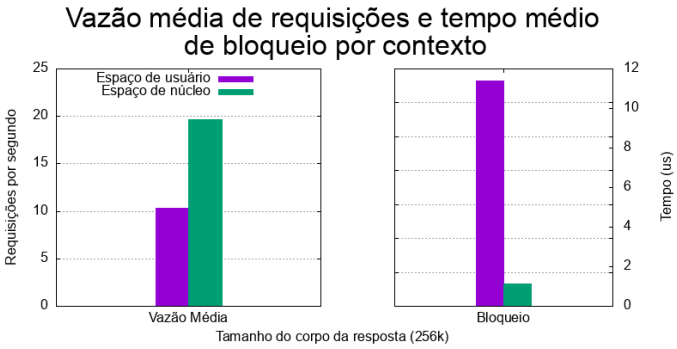


Figura 4. Média harmônica da vazão de requisições por segundo e de tempo total para bloqueio entre bloqueio feito em espaço de usuário e espaço de núcleo.

### B. Tempo de CPU para executar bloqueio

No gráfico à direita da figura 4 nota-se que a implementação em espaço de núcleo gasta **26,4% menos tempo de CPU** para bloquear pacotes do que a implementação em espaço de usuário, utilizando apenas 8 microssegundos para efetuar o bloqueio de um pacote.

### C. Duração das requisições

A imagem 5 mostra a duração média (harmônica) computada durante o experimento. Note que a duração média de requisições sem controle de quotas é menor do que para o controle feito em espaço de usuário. Isso ocorre, pois ao fazer controle de quotas, a aplicação precisa executar mais código.

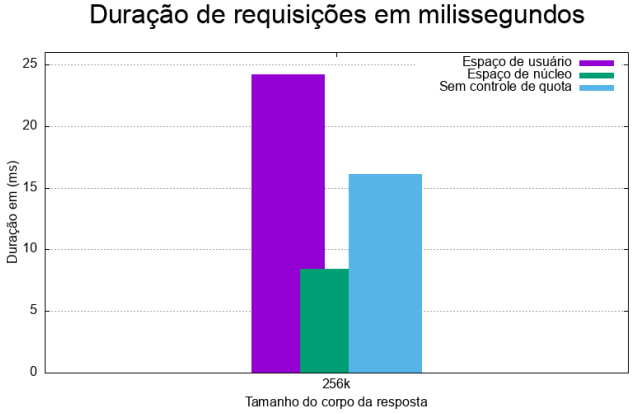


Figura 5. Média harmônica da duração das requisições entre bloqueio feito em espaço de usuário e espaço de núcleo e sem bloqueio algum (redução de 65%)

Foi observado uma **redução de 65%** na **duração média** das requisições HTTPS quando feito o controle de quotas no espaço de núcleo comparado ao espaço de usuário.

### D. Comparação de volume de chamadas de sistema

A tabela VIII-D mostra o número de chamadas de sistema feitos durante o experimento paralelo com e sem o uso do *eQUIC*. Notou-se que o uso do *eQUIC* **reduziu em 89% o número de chamadas de sistema**.

| Programa | Número de syscalls | Número de erros em syscalls | Tempo ocupado em syscalls (s) |
|---|---|---|---|
| Sem eQUIC | 77551 | 562 | 2.936784 |
| Com eQUIC | 8178 | 26 | 0.312806 |

Tabela I
VALORES COLETADOS COM O VOLUME DE CHAMADAS DE SISTEMA FEITAS DURANTE O EXPERIMENTO PARALELO.

Verifica-se também uma redução no tempo gasto ocupado em executar as chamadas de sistema assim como no número de erros. Uma proporção na ordem de 10 vezes menos tempo e erros. Isso acontece, pois com o bloqueio sendo feito logo ao receber o pacote, poupa-se o sistema operacional à sobrecarga de trabalho computacional necessária para que a aplicação em espaço de usuário faça esse mesmo bloqueio.

### E. Percentis de CPU e Memória

Foram medidos os percentis de uso de CPU e memória durante o experimento paralelo. Ao utilizar o *eQUIC*, 90% do tempo do experimento o sistema utilizou menos de 40% do CPU. Por outro lado, sem o *eQUIC*, 80% do tempo do experimento o sistema utilizou 90% do recurso CPU. A figura 6 apresenta os gráficos das funções acumuladas das distribuição desse recurso.

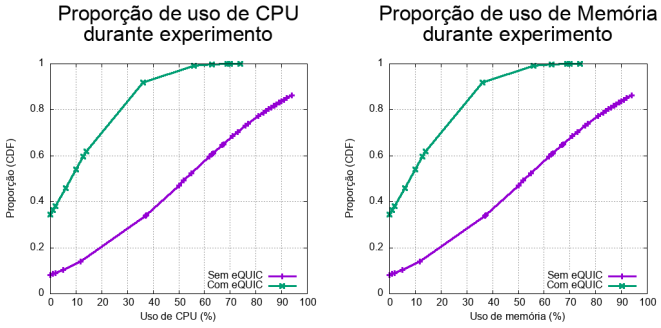


Figura 6. Funções acumuladas de distribuição do uso dos recursos computacionais CPU e Memória

Um comportamento semelhante pôde ser visto para o recurso Memória. 90% do tempo do experimento utilizando *eQUIC* consumiu cerca de 30% da memória disponível enquanto que sem *eQUIC* esse valor é 90%.

## IX. TRABALHOS FUTUROS

O presente trabalho não descriptografa o pacote QUIC em espaço de núcleo. Toda informação interna do protocolo precisa ser informada ao núcleo através de mapas eBPF. Uma evolução deste trabalho seria fazer (*offload*) da computação criptográfica do protocolo para o espaço de núcleo de modo a reduzir ainda mais a troca de contexto e custo de CPU em espaço de usuário.

A biblioteca KTLS (*Kernel TLS*) [27], originalmente proposta pelo trabalho de Watson [28], permite processar a criptografia e descriptografia simétrica do protocolo TLS dentro do núcleo do sistema operacional [29].

No entanto, esta biblioteca ainda não suporta UDP, o que é um empecilho para sua utilização com o protocolo QUIC. Contudo, o QUIC *Gateway* poderia levar para o núcleo outras tarefas como controle de taxa (*rate limit*), autorização, autenticação e até mesmo procuração (*proxy*) centralizado para um conjunto de aplicações; não apenas um único servidor.

Outra perspectiva de evolução é que o programa em espaço de usuário atua como plano de controle e o em núcleo como plano de dados programável, o que permitiria criar uma solução em Redes Definidas por Software [30].

## X. CONCLUSÃO

Este trabalho apresenta uma análise prática do *offload* de tarefas executadas em espaço de usuário para o espaço de núcleo. Utilizando como objeto de pesquisa o protocolo QUIC em um ambiente de experimentação, substituiu-se a pilha de protocolos Internet, como TCP+TLS+HTTP, de modo a implementar um servidor HTTPS.

Criou-se um serviço que atua como *Gateway* em um sistema distribuído entre espaço de usuário e espaço de núcleo que tem como objetivo controlar quais pacotes estão autorizados a prosseguir pela pilha de protocolos do sistema operacional em direção a aplicação. O eQUIC Gateway, em seus resultados computados, aumentou a vazão de pacotes em 30,9%, também reduziu em 65% a duração média em requisições HTTPS, reduziu em 26,4% o tempo de CPU necessário para bloquear-se pacotes e fez 89% menos chamadas de sistema (*syscall*).

Foram discutidos os trabalhos futuros e como o eQUIC *Gateway* pode evoluir para ser um plano de dados programável de tempo real para o protocolo QUIC em espaço de núcleo. O código fonte do eQUIC está disponível no Github [31].

## AGRADECIMENTOS

Os autores gostariam de agradecer às agências de fomento de pesquisa CNPq, FAPESP 2018/23085-5 e FAPEMIG pelo suporte financeiro.

## REFERÊNCIAS

[1] QUIC-Draft, *QUIC protocol Draft*, 2020. [Online]. Available: https://tools.ietf.org/html/draft-ietf-quic-transport-29

[2] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, and Z. Shi, "The quic transport protocol: Design and internet-scale deployment," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 183–196. [Online]. Available: https://doi.org/10.1145/3098822.3098842

[3] J. Rüth, I. Poese, C. Dietzel, and O. Hohlfeld, "A first look at quic in the wild," in *Passive and Active Measurement*, R. Beverly, G. Smaragdakis, and A. Feldmann, Eds. Cham: Springer International Publishing, 2018, pp. 255–268.

[4] Y. Moon, S. Lee, M. A. Jamshed, and K. Park, "Acceltcp: Accelerating network applications with stateful TCP offloading," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 77–92. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/moon

[5] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, pp. 74–80, 2013. [Online]. Available: http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext

[6] A. M. Kakhki, S. Jero, D. Choffnes, C. Nita-Rotaru, and A. Mislove, "Taking a long look at quic: An approach for rigorous evaluation of rapidly evolving transport protocols," in *Proceedings of the 2017 Internet Measurement Conference*, ser. IMC '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 290–303. [Online]. Available: https://doi.org/10.1145/3131365.3131368

[7] M. Palmer, T. Krüger, B. Chandrasekaran, and A. Feldmann, "The quic fix for optimal video streaming," in *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, ser. EPIQ'18. New York, NY, USA: Association for Computing Machinery, 2018, p. 43–49. [Online]. Available: https://doi.org/10.1145/3284850.3284857

[8] V. Sivakumar, T. Rocktäschel, A. H. Miller, H. Küttler, N. Nardelli, M. Rabbat, J. Pineau, and S. Riedel, "Mvfst-rl: An asynchronous rl framework for congestion control with delayed actions," *NeurIPS Workshop on Machine Learning for Systems*, 2019. [Online]. Available: https://arxiv.org/abs/1910.04054

[9] M. A. M. Vieira, M. S. Castanho, R. D. G. Pacífico, E. R. S. Santos, E. P. M. Câmara Júnior, and L. F. M. Vieira, "Processamento Rápido de Pacotes com eBPF e XDP," in *Minicursos do XXXVII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuıdos (SBRC)*. Porto Alegre, RS, Brasil: SBC, May 2019.

[10] eBPF, *eBPF - extended Berkeley Packet Filter*, 2020. [Online]. Available: https://prototype-kernel.readthedocs.io/en/latest/bpf/

[11] M. A. M. Vieira, M. S. Castanho, R. D. G. Pacífico, E. R. S. Santos, E. P. M. C. Júnior, and L. F. M. Vieira, "Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications," *ACM Comput. Surv.*, vol. 53, no. 1, Feb. 2020. [Online]. Available: https://doi.org/10.1145/3371038

[12] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The express data path: Fast programmable packet processing in the operating system kernel," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 54–66. [Online]. Available: https://doi.org/10.1145/3281411.3281443

[13] S. McCanne and V. Jacobson, "The bsd packet filter: A new architecture for user-level packet capture," in *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, ser. USENIX'93. USA: USENIX Association, 1993, p. 2.

[14] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, p. 87–95, Jul. 2014. [Online]. Available: https://doi.org/10.1145/2656877.2656890

[15] X. Yang, L. Eggert, J. Ott, S. Uhlig, Z. Sun, and G. Antichi, "Making quic quicker with nic offload," in *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, ser. EPIQ '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 21–27. [Online]. Available: https://doi.org/10.1145/3405796.3405827

[16] P. Wang, C. Bianco, J. Riihijärvi, and M. Petrova, "Implementation and performance evaluation of the quic protocol in linux kernel," in *Proceedings of the 21st ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, ser. MSWIM '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 227–234. [Online]. Available: https://doi.org/10.1145/3242102.3242106

[17] Q. De Coninck, F. Michel, M. Piraux, F. Rochet, T. Given-Wilson, A. Legay, O. Pereira, and O. Bonaventure, "Pluginizing quic," in *Proceedings of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 59–74. [Online]. Available: https://doi.org/10.1145/3341302.3342078

[18] uBPF, *uBPF - User-space Berkeley Packet Filter*, 2020. [Online]. Available: https://github.com/iovisor/ubpf

[19] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The operating system is the control plane," *ACM Trans. Comput. Syst.*, vol. 33, no. 4, Nov. 2015. [Online]. Available: https://doi.org/10.1145/2812806

[20] DPDK, *Data Plane Development Kit*, 2021. [Online]. Available: https://www.dpdk.org/

[21] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 361–378. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/ousterhout

[22] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay, "Caladan: Mitigating interference at microsecond timescales," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 281–297. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/fried

[23] R. Pacífico, L. Duarte, M. Castanho, J. Miranda Nacif, and M. A. M. Vieira, "Sistema de processamento de pacotes serverless," in *Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuıdos (SBRC)*, 12 2020, pp. 183–196.

[24] A. G. Vieira, G. Pantuza, J. H. F. Freire, L. F. S. Duarte, R. D. G. Pacífico, G. H. A. Pereira, M. A. M. Vieira, L. F. M. Vieira, and J. A. M. Nacif, "Computação Serverless: Conceitos, Aplicações e Desafios," in *Minicursos do XXXVIII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuıdos (SBRC)*. Rio de Janeiro, RJ, Brasil: SBC, May 2020.

[25] LiteSpeed, *LiteSpeed QUIC (LSQUIC) Library*, 2021. [Online]. Available: https://github.com/litespeedtech/lsquic

[26] Docker, *Containerization of applications with Docker*, 2021. [Online]. Available: https://github.com/docker

[27] L. Kernel, *Kernel TLS documentation*, https://www.kernel.org/doc/html/latest/networking/tls.html#kernel-tls, 2020.

[28] D. Watson, "Ktls: Linux kernel transport layer security," *Proposal by Facebook Engineer*, 2016.

[29] ——, *TLS in the kernel*, 2015. [Online]. Available: https://lwn.net/Articles/666509/

[30] G. Pantuza, F. Sampaio, L. F. M. Vieira, D. Guedes, and M. A. M. Vieira, "Network management through graphs in software defined networks," in *10th International Conference on Network and Service Management (CNSM) and Workshop*, 2014, pp. 400–405.

[31] G. Pantuza, *eQUIC source code*, 2021. [Online]. Available: https://github.com/pantuza/equic