Análise de desempenho do nsQUIC: um Módulo para Simulação do Protocolo QUIC

Diego de Araújo Martinez Camarinha

Dissertação apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
obtenção do título
de
Mestre em Ciências

Programa: Ciência da Computação

Orientador: Prof. Dr. Daniel Macêdo Batista

São Paulo, julho de 2018

Análise de desempenho do nsQUIC: um Módulo para Simulação do Protocolo QUIC

Esta versão da dissertação contém as correções e alterações sugeridas pela Comissão Julgadora durante a defesa da versão original do trabalho, realizada em 23/08/2018. Uma cópia da versão original está disponível no Instituto de Matemática e Estatística da Universidade de São Paulo.

Comissão Julgadora:

- Prof. Dr. Daniel Macêdo Batista (orientador) - IME-USP
- Prof. Dr. Marco Dimas Gubitoso - IME-USP
- Prof. Dr. Fernando Frota Redigolo - Poli-USP

Resumo

CAMARINHA, D. A. M. **Análise de desempenho do nsQUIC: um Módulo para Simulação do Protocolo QUIC**. 2018. Dissertação de Mestrado - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2018.

Várias características da Internet mudaram drasticamente desde que o TCP foi criado, como o maior compartilhamento de recursos devido à maior quantidade de usuários, maior largura de banda disponível, a existência de muitas conexões que podem percorrer longas distâncias e a ubiquidade das redes sem fio. Confrontado com essas novas características, o TCP apresenta diversas limitações. Dentre elas estão a subutilização da rede quando a largura de banda é da ordem de centenas de Gbps, o favorecimento de conexões que possuem pouco atraso (poucas dezenas de milisegundos), a incapacidade de distinção de causas de perdas de pacote e a demora para estabelecimento de conexões seguras (até 3 RTTs). Nesse contexto, com o objetivo de tornar o transporte de dados na Internet mais rápido e eficiente, a Google desenvolveu o protocolo QUIC. O QUIC propõe diversos avanços em relação aos protocolos existentes, como um novo mecanismo para estabelecimento de conexão e controle de congestionamento otimizado. Resultados apresentados pela Google mostraram claro ganho de desempenho em relação ao TCP, justificando o trabalho de tornar o QUIC um padrão IETF da Internet. Porém, esses resultados são impossíveis de serem verificados porque nos relatórios divulgados não há informação suficiente para que os cenários de teste sejam reproduzidos e porque é implausível possuir a mesma infraestrutura para os testes que a Google tem. Neste trabalho, avaliamos o desempenho do protocolo QUIC em diversos cenários de rede, comparando-o com o desempenho de várias implementações do TCP, principalmente o CUBIC. Diferente do realizado na literatura, todos os cenários utilizados são bem descritos, permitindo a reprodutibilidade dos experimentos. Além disso, para a realização dos experimentos foi criado um novo módulo que implementa o QUIC no simulador de redes NS-3. Este módulo está disponível como software livre, permitindo que outros pesquisadores usem o módulo para replicar e verificar nossos experimentos e para criarem novos experimentos de forma reprodutível. Ademais, eles também podem usar o módulo como uma ferramenta para avaliar, de maneira rápida, o comportamento de novas técnicas dentro do protocolo.

Palavras-chave: QUIC, TCP, protocolo de transporte, análise de desempenho.

Abstract

CAMARINHA, D. A. M. **Performance Analysis of nsQUIC: a Simulation Module for the QUIC Protocol**. 2018. Masters Thesis - Institute of Mathematics and Statistics, Sao Paulo University, Sao Paulo, 2018.

Many characteristics of the Internet have drastically changed since TCP was created such as the increase on resource sharing due to a larger number of Internet users, the growth of available bandwidth, the existence of many connections that may travel long distances and the ubiquity of wireless networks. When faced with those new characteristics, TCP showed severe limitations. Among them are network underutilization in high bandwidth environments of hundreds of Gbps, favoring of connections with small delays (few tens of milliseconds), incapacity of distinguishing packet loss causes and high delays for establishing secure connections (up to 3 RTTs). In this context, with the goal of making Internet data transport faster and more efficient, Google has developed the QUIC protocol. QUIC proposes many advances compared to existing protocols, such as a new mechanism for establishing connections and an optimized congestion control algorithm. Google has reported results indicating that QUIC performs better than TCP, justifying the work on making QUIC an IETF Internet standard. However, those results cannot be verified because on the published reports there is not enough information to reproduce the test scenarios and it is implausible to have the same test infrastructure Google has. In this work, we evaluate QUIC's performance in a number of network scenarios, comparing it with the performance of different TCP flavours, specially TCP CUBIC. Unlike other works in the literature, all scenarios are well described, enabling experiment replicability. Furthermore, to run experiments we created a new module that implements QUIC on the network simulator NS-3. The module is available as free software, allowing other researchers to use it to reproduce and verify our experiments and to create new ones in a replicable way. Additionally, they can use the module as a tool to quickly assess the behaviour of new techniques in the protocol.

Keywords: QUIC, TCP, transport protocol, performance analysis.

Sumário

Lista de Abreviaturas

ACK	*Acknowledgement*
AIMD	*Additive-Increase Multiplicative-Decrease*
API	*Application Programming Interface*
BBR	*Bottleneck Bandwidth and Round-trip Propagation Time*
CHLO	*Client Hello*
CID	*Connection Identifier*
DTLS	*Datagram Transport Layer Security*
FEC	*Forward Error Correction*
HTTP	*Hypertext Transfer Protocol*
IETF	*Internet Engineering Task Force*
LHC	*Large Hadron Collider*
LIGO	*Laser Interferometer Gravitational-Wave Observatory*
MPTCP	*Multipath TCP*
MSS	*Maximum Segment Size*
NAT	*Network Address Translation*
NS-3	*Network Simulator 3*
QUIC	*Quick UDP Internet Connections*
REJ	*Rejection*
RFC	*Request For Comments*
RST	*Reset*
RTT	*Round-Trip Time*
SACK	*Selective Acknowledgement*
SCTP	*Stream Control Transmission Protocol*
SHLO	*Server Hello*
TCP	*Transmission Control Protocol*
TLP	*Tail Loss Probe*
TLS	*Transport Layer Security*
UDP	*User Datagram Protocol*
XOR	*Exclusive Or*

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

O número de usuários da Internet está aumentando rapidamente. Segundo os últimos dados publicados pelo *Internet World Stats*, no final de dezembro de 2017, esse número era de mais de 4,1 bilhões de pessoas, representando 54,4% da população mundial. Na maioria das regiões do planeta, mais da metade da população tem acesso à Internet. O crescimento do número de usuários no período entre os anos 2000 e 2018 foi de 1052%. Além disso, nesse mesmo período, o crescimento no Brasil foi de 2854,5%, com um total de 70,7% da população com acesso à Internet [IWS].

Ao mesmo tempo, a quantidade de conteúdo produzido pelos usuários e distribuído pela Internet vem crescendo a uma velocidade surpreendente. Esse fato foi observado pela Cisco®, em um relatório que apresenta previsões feitas pela empresa sobre o tráfego global de IP para 2021 [Ind17]. Dentre as previsões mais interessantes estão:

- O tráfego IP global anual chegará a 3,3 *zettabytes*;
- O tráfego IP de *smartphones* ultrapassará o tráfego de computadores;
- O tráfego de dispositivos sem fio e móveis será responsável por aproximadamente 63% do tráfego IP;
- A velocidade média de banda larga quase dobrará até 2021: será de 53Mbps enquanto que, em 2014, era de 27,5Mbps;
- Globalmente, o tráfego de vídeo será quase 82% de todo o tráfego IP e,
- Haverá aproximadamente 1,75 conexões entre máquinas (M2M) para cada membro da população global.

Essas previsões mostram aspectos interessantes sobre a heterogeneidade de conexões na Internet, sobre a diversidade de aplicações existentes e sobre a quantidade de dados em circulação na rede.

Uma parte significativa da quantidade de dados movimentada pela rede diariamente é proveniente de experimentos científicos. O LIGO (*Laser Interferometer Gravitational-Wave Observatory*) [LIGb], observatório que recentemente detectou a existência de ondas gravitacionais [LIGa], movimenta terabytes de dados todos os dias [LIGc]. O LHC (*Large Hadron Collider*) [LHC], em 2011, já captava 1 petabyte de dados por sensor [Bir11] que precisava ser movimentado para processamento distribuído ao redor do mundo. Nesse tipo de transferência, tipicamente a rede possui enlaces de alta capacidade (da ordem de dezenas de Gbps), porém com alto atraso porque os dados percorrem distâncias continentais.

O *download* de vídeos e *streaming* atingem um público-alvo diferente. Nesse caso, os usuários normalmente estão conectados a redes com capacidade média (dezenas de Mbps) e com pouco atraso.

Por fim, no contexto de Internet das coisas, os dispositivos possuem pouca capacidade de processamento, estão conectados a redes com alta perda de pacotes, com taxa de transmissão restrita e pouco atraso.

Apesar das diferentes características da rede e das aplicações, o protocolo predominantemente utilizado para transporte é o TCP (*Transmission Control Protocol*). Porém, esse protocolo possui limitações em diversas situações. Seu mecanismo de detecção de erros não sabe diferenciar tipos de erro e assume que sempre que um pacote é perdido a razão é congestionamento. A reação do TCP quando detecta congestionamento é reduzir a velocidade de transmissão. Esse cenário nem sempre é verdade para redes sem fio, nas quais perdas de pacotes ocorrem frequentemente por conta do meio de transmissão. Outros problemas relacionados a redes sem fio são o gasto de energia e o algoritmo de recuperação de erros do protocolo [TM02] [XPMS01]. O algoritmo de partida lenta do TCP pode demorar vários RTTs (*round-trip times*) até aproveitar toda a capacidade da rede. Para transações pequenas e redes com alta capacidade, o protocolo subutilizará os recursos disponíveis [WPY+04]. O tamanho máximo dos pacotes (64KB) pode interferir na utilização ótima de recursos da rede e no desempenho de transferências. O intervalo de número de sequência (32 bits) pode trazer complicações no reconhecimento de pacotes em redes de alta capacidade, pois os números disponíveis podem esgotar antes mesmo de o primeiro pacote ACK (*Acknowledgement* - que indica reconhecimento de um pacote) ser recebido. O TCP também é injusto com pacotes que possuem alto atraso de transmissão [LM97]. Além disso, o TCP é implementado pelos sistemas operacionais, ou seja, não é um programa que roda no espaço de usuário. Isso quer dizer que qualquer tipo de melhoria no protocolo pode demorar para realmente ser validada em redes reais e implantadas na infraestrutura atual pois todos os sistemas precisam ser atualizados.

No contexto de otimização dos recursos da rede e, principalmente, de maior velocidade na transmissão de dados, a Google propôs o QUIC (*Quick UDP Internet Connections*) [Ros], um novo protocolo de transporte para a Internet. Esse protocolo usa multiplexação e é construído sobre o UDP (*User Datagram Protocol*). Seu desenvolvimento e implementação começaram em 2012 e, desde então, o protocolo evoluiu e se tornou relevante para a Internet. Por exemplo, ainda no começo de 2015 já era possível utilizá-lo como uma extensão do navegador *Chromium* e metade das requisições para os servidores da Google eram servidas usando o QUIC [WHS15].

Ainda em 2015, a Google divulgou resultados de experimentos que foram realizados com o QUIC rodando em produção na Internet [WHS15]. Esses experimentos apontaram que o QUIC apresenta vantagens significativas em relação ao TCP: desempenho 3% melhor para carregar uma página de busca da Google e 30% menos *rebuffers* em vídeos do YouTube. Na metade de 2016, um grupo de trabalho da IETF (*Internet Engineering Task Force*) foi criado para que o QUIC se torne um novo padrão da Internet. Em outro estudo mais recente [LRW+17], de 2017, a Google confirma os ganhos de desempenho do QUIC bem como dimensiona seu espaço na Internet atualmente: o QUIC já é responsável por 7% de todo o tráfego na Internet. Em 2018, o QUIC já vem habilitado por padrão no *Chromium* e todas as requisições feitas por ele para servidores da Google são respondidas também por QUIC.

Este trabalho estuda diferentes métodos de análise de desempenho de protocolos de transporte da Internet e usa o QUIC como objeto de estudo. A análise do QUIC, que representa a principal contribuição desta dissertação, é realizada com a utilização de um novo módulo para o simulador de redes NS-3.

1.1 Motivação

Apesar dos resultados reportados pela Google serem animadores, os relatórios divulgados não possuem nenhum tipo de informação sobre a configuração dos experimentos: número de clientes e servidores, quantidade de dados enviados, largura de banda e atraso dos enlaces, como as métricas foram calculadas, quais as limitações dos experimentos, etc. Sem isso, não é possível reproduzir os experimentos para verificar sua veracidade e validade. Ademais, mesmo que tivéssemos acesso a todas essas informações, a Google possui infraestrutura própria com enlaces dedicados e implementação otimizada (de código fechado) do servidor, tornando implausível a reprodução exata dos experimentos.

Analisar o desempenho do QUIC por meio de experimentos reprodutíveis é de vital importância

para a detecção rápida de gargalos e de técnicas que não se comportam da maneira esperada. Essa análise pode servir de base para tomadas de decisão que poderão alterar a especificação do protocolo quando ele finalmente se tornar um padrão IETF da Internet.

Por fim, vale lembrar que o desempenho do protocolo de transporte impacta diretamente na qualidade de experiência dos usuários. Por isso, os resultados desta análise de desempenho do QUIC é relevante para que desenvolvedores possam tomar uma decisão embasada e confiável sobre se vale a pena substituir o TCP pelo QUIC.

1.2 Objetivos

Os principais objetivos deste trabalho foram:

- Estudar os principais métodos de avaliação de desempenho: modelagem, simulação e medição;
- Estudar protocolos de transporte e como são feitas análises de desempenho nesse contexto;
- Aplicar o que foi aprendido analisando o desempenho do protocolo QUIC, de forma que esta análise seja facilmente replicável e,
- Deixar como legado uma maneira simples de estender a análise do protocolo QUIC tanto para como o protocolo está definido atualmente quanto para abranger futuras mudanças.

1.3 Contribuições

As principais contribuições deste trabalho são:

1. Análise inédita de desempenho do protocolo QUIC por meio de simulações no NS-3 que também serve como base para validação de trabalhos de medição ou modelagem analítica e,
2. Implementação de um módulo do QUIC para o NS-3.

1.4 Organização do Trabalho

O Capítulo 2 está organizado de forma a explicar a base de conceitos necessária para a compreensão deste trabalho, justificando a escolha por simulação e pelo QUIC. Descrevemos os principais métodos de avaliação de desempenho, explicamos características fundamentais do TCP e seus problemas para, em seguida, fazermos uma comparação com as técnicas empregadas pelo QUIC para resolver tais problemas. Por fim, também apresentamos outro protocolo que, assim como o QUIC, tem como objetivo substituir o TCP.

O Capítulo 3 apresenta outros trabalhos relacionados com o nosso, que visam avaliar o desempenho de protocolos de transporte por meio de modelagens, simulações ou medições.

O Capítulo 4 detalha a metodologia que utilizamos para integrar o código do QUIC no NS-3, bem como explica decisões técnicas, expõe os obstáculos encontrados e as soluções para superá-los.

O Capítulo 5 descreve as simulações que realizamos, detalhando as topologias de rede, fatores que variamos e quais métricas calculamos. Esse capítulo também expõe e comenta os resultados obtidos nas simulações e explica como os validamos.

Por fim, o Capítulo 6 conclui o trabalho, discute suas limitações e propõe trabalhos futuros.

Capítulo 2

Conceitos

Neste capítulo, explicamos conceitos essenciais para a compreensão deste trabalho. Definimos e descrevemos os principais métodos de avaliação de desempenho: modelagem, simulação e medição. Exemplificamos como cada um deles pode ser aplicado no contexto de redes de computadores e justificamos a escolha por simulação no nosso trabalho. Em seguida, apontamos as principais características, variantes e problemas do estabelecimento de conexão e controle de congestionamento no TCP porque esses dois aspectos são os que mais impactam o desempenho do transporte de dados entre servidores e clientes. Detalhamos o funcionamento do QUIC, destacando as técnicas mais relevantes que o diferenciam do funcionamento do TCP para resolver seus problemas e, finalmente, apresentamos brevemente o SCTP, outro protocolo de transporte cujo objetivo também é substituir o TCP.

2.1 Análise de Desempenho

Desempenho é uma métrica crucial que guia o desenvolvimento e o uso de sistemas de computadores. Um sistema de computador pode ser um *hardware*, como uma CPU, um *software*, como um banco de dados, ou reunir os dois, como uma rede de computadores. Dependendo do sistema, seu desempenho pode ser medido de diferentes maneiras. Por exemplo, o desempenho de uma CPU pode ser medido como o número de instruções por ciclo que ela consegue executar. Já para um banco de dados e para uma rede de computadores, o desempenho pode ser medido em termos do número de transações por segundo e pelo tempo de transferência de dados de um computador para outro, respectivamente. Independentemente do sistema e da métrica que será utilizada, a análise de desempenho é de suma importância para se tomar a decisão que maximize o desempenho e minimize os custos. Por exemplo, a análise de desempenho é necessária quando se quer comparar sistemas ou algoritmos para se decidir qual é o melhor para uma determinada situação. No contexto específico deste trabalho, no qual há discussão na comunidade científica sobre se vale a pena substituir o TCP pelo QUIC, analisar e comparar o desempenho de ambos os protocolos fornece base imprescindível para uma tomada de decisão melhor informada.

Existem 3 técnicas para realizar análise de desempenho: modelagem analítica, simulação e medição [Jai90]. Na modelagem analítica, usamos fórmulas matemáticas que descrevem o sistema e as mudanças que podem ocorrer nele. Por exemplo, o controle de congestionamento de protocolos de transporte é definido por fórmulas matemáticas que descrevem como a janela de congestionamento deve se comportar quando mudanças no sistema, como uma perda de pacote, ocorrem. Em simulações, sistemas são replicados por *software*. Essas réplicas são definidas por meio de modelos matemáticos que possuem suposições e restrições em sua construção. Nesse sentido, simulações capturam mais características do sistema e oferecem maior nível de detalhe na avaliação se comparadas a modelagens analíticas. Porém, apesar de simulações serem importantes para prever o desempenho e comparar alternativas, seus resultados devem ser analisados com cuidado para que as conclusões não sejam precipitadas. Por exemplo, simulações podem ser usadas para prever o desempenho de transferências de dados em redes móveis mas o modelo de mobilidade usado impõe limitações na

simulação que impactam diretamente em seus resultados. No contexto de computação, existem 4 tipos de simulação: emulação, simulação de Monte Carlo, simulações orientadas a rastros e simulações de eventos discretos.

- **Emulação** é uma simulação que usa *hardware* ou *firmware*. Por exemplo, um emulador de um processador imita o conjunto de instruções de um processador em outro;
- **Simulação de Monte Carlo** utiliza amostragem aleatória repetidas vezes para se poder estudar os resultados de processos complexos. Esse tipo de simulação tem inúmeras aplicações como, por exemplo, na aproximação númerica de integrais multidimensionais, em simulações físicas, no mercado financeiro e em gerenciamento de projetos;
- **Simulação orientada a rastros** usa registros de eventos cronologicamente ordenados como parâmetro de entrada da simulação. Os rastros podem ser tanto sintéticos quanto obtidos de sistemas reais. Por exemplo, rastros de uso de CPU podem ser utilizados para se comparar algoritmos de escalonamento e,
- **Simulação de eventos discretos** diz respeito ao uso de modelos que capturam estados estáticos de um sistema. Nesse tipo de simulação, uma mudança de estado do sistema se dá por meio de um evento. Por exemplo, em um simulador de redes de computadores, o envio de dados de uma ponta a outra é um evento e o recebimento desses dados é outro.

Medições, por sua vez, são uma forma de análise de desempenho em que os experimentos são realizados em sistemas reais. Por exemplo, é possível calcular o tempo total de carregamento de uma página web para se avaliar se determinado protocolo de aplicação tem o desempenho esperado.

Escolher qual técnica utilizar pode ser difícil, porém a Tabela 2.1, adaptada de [Jai90], pode ajudar nessa escolha. Nela, os critérios estão listados em ordem de importância. Todos eles são explicados a seguir.

Critério	**Modelagem Analítica**	**Simulação**	**Medição**
1. Estágio	Qualquer	Qualquer	Pós prototipação
2. Tempo requerido	Pequeno	Médio	Varia
3. Ferramentas	Analistas	Linguagens de programação	Instrumentação
4. Acurácia	Baixa	Moderada	Varia
5. Avaliação	Fácil	Moderada	Difícil
6. Custo	Baixo	Médio	Alto
7. Capacidade de convencimento	Baixa	Média	Alta

Tabela 2.1: *Critérios para escolha de qual técnica de análise de desempenho utilizar. A lista está ordenada em ordem de importância. Tabela adaptada de [Jai90].*

1. O estágio em que o sistema está é o critério mais importante para a escolha de qual técnica utilizar. Se o sistema ainda não existe ou ainda é um protótipo, medição não é uma opção;
2. Se o tempo disponível para a análise é curto, modelagem analítica é uma boa opção. Simulações podem demorar um bom tempo e medições podem variar por conta de erros externos e imprevistos;
3. As ferramentas disponíveis para cada técnica impõem uma restrição em quem realizará a análise. Normalmente será escolhida a técnica cujas ferramentas são mais familiares para essa pessoa;

4. A acurácia desejada também influencia na escolha. Modelagem analítica necessita de tantas suposições e simplificações que os resultados normalmente não serão acurados. Simulações capturam mais características do sistema e são mais detalhadas, fazendo com que fiquem mais próximas da realidade. Medições podem não ser acuradas caso os parâmetros usados não representem a gama de variáveis encontradas na realidade. É importante ressaltar que acurácia não implica em corretude da análise;

5. A avaliação do impacto de alterações nos parâmetros do sistema para comparação de desempenho também é determinante. Normalmente, em medições é difícil saber se um ganho ou uma perda de desempenho é resultado de uma configuração específica ou se é impacto de alguma mudança no ambiente. Em modelagem analítica e simulações, geralmente é mais fácil compreender os efeitos dos parâmetros do experimento e sua interações;

6. Dentre as 3 técnicas, a medição é a técnica mais custosa. Ela necessita de equipamentos reais, por exemplo. Por ser custosa, normalmente medições são substituídas por simulações. Modelagem analítica é a técnica mais barata;

7. Por fim, a capacidade de convencimento de medições por vezes faz valer a pena seu custo e trabalho.

Neste trabalho, escolhemos utilizar simulações principalmente pelo alto custo que teríamos se optássemos por medições: precisaríamos ter o controle de pelo menos um servidor externo conectado à Internet e de um roteador para podermos experimentar alguns parâmetros como a largura de banda e atraso adicional na transmissão de pacotes. Outro fator decisivo foi a possibilidade de contribuir com a comunidade científica ao viabilizar uma maneira nova e de baixo custo de realizar experimentos reprodutíveis com o protocolo QUIC.

Finalmente, é importante não confiar nos resultados de uma técnica por si só. É necessário algum tipo de validação. Para isso, podemos usar os resultados de duas técnicas para validar um ao outro. Por exemplo, podemos usar os resultados de simulações para validar os resultados de medições e vice-versa. Neste trabalho, validamos os resultados das nossas simulações usando os resultados de medições publicados em [KJC+17].

2.2 TCP

O *Transmission Control Protocol*, apesar de ser o protocolo de transporte mais utilizado na Internet, foi desenvolvido muito antes dela. Ainda no início da década de 1970, os pesquisadores Vinton Cerf e Robert Kahn inventaram o primeiro protocolo que propunha unificar a intercomunicação entre computadores, na época chamado de TCP/IP. Mais tarde, esse protocolo foi dividido em dois, o TCP e o IP. Em 1981, o TCP foi especificado na RFC 793 [Pos81]. Já naquela época, Cerf e Kahn anteciparam que havia a necessidade de um protocolo que fosse abrangente o suficiente para dar suporte a aplicações que ainda seriam criadas e para permitir interoperabilidade entre *hosts* e protocolos de enlace diferentes. As características básicas do protocolo, ser orientado a conexões e fornecer transporte confiável dos dados, foram definidas nesse período. Ser orientado a conexões significa que, antes que uma aplicação possa enviar dados para outra, elas precisam primeiro trocar segmentos preliminares para inicializar alguns parâmetros que serão usados na transferência de dados. Portanto, a conexão entre dois sistemas é feita de forma lógica por meio desses parâmetros. Uma conexão TCP é sempre de ponta-a-ponta, entre um único remetente e destinatário. Ela também é dita *full-duplex*, ou seja, as duas aplicações da conexão podem enviar dados uma para a outra simultaneamente. Para fornecer transporte confiável dos dados, o TCP usa como base os princípios de detecção de erros, retransmissões, reconhecimento cumulativo dos segmentos recebidos, temporizadores e números de sequência. Um número de sequência identifica um segmento. Reconhecimento cumulativo dos segmentos recebidos significa que um *host* indicará que recebeu todos os segmentos até o primeiro ausente na comunicação. Por exemplo, suponha que um *host* **A**

recebeu de um *host* **B** os segmentos 1 e 3. Para recriar o fluxo de dados que **B** está enviando, **A** precisa do segmento 2. Portanto, ele indicará que recebeu apenas o segmento 1. Temporizadores são utilizados para retransmitir os segmentos emitidos que passam determinado tempo (*timeout*) sem serem reconhecidos pelo destinatário. A detecção de erros é feita usando o esquema de *checksum*.

No restante desta seção, explicaremos alguns detalhes de funcionamento do TCP que impactam diretamente em seu desempenho, para na seção seguinte explicar como o QUIC propõe soluções diferentes para eles.

2.2.1 Estabelecimento de Conexão

Como foi mencionado anteriormente, o TCP é orientado a conexões e segmentos preliminares precisam ser trocados antes que um cliente possa enviar dados para um servidor. A Figura 2.1 mostra como a comunicação é feita nesse processo. Quando um cliente quer iniciar uma conexão com o servidor, ele primeiro envia um segmento SYN para o servidor. Ao receber esse segmento, o servidor responde com um segmento SYNACK que indica tanto o recebimento do segmento SYN do cliente quanto o aceite do pedido de conexão. O cliente, então, envia um segmento ACK para indicar que está ciente de que a conexão está estabelecida. Ele também pode enviar dados para o servidor já nesse segmento. Esse processo de estabelecimento de conexão, por necessitar da troca de 3 segmentos, é denominado de *3-way handshake*. Note que, dessa maneira, o TCP impõe a restrição de um cliente necessariamente ter de esperar 1 RTT, tempo passado entre o envio de um pacote e o recebimento da confirmação que ele alcançou o destino (ACK), para poder começar a enviar dados para o servidor.

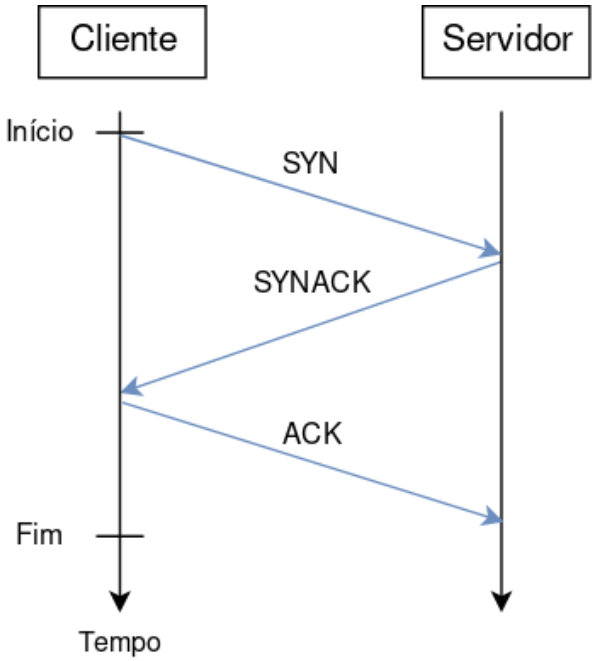

Figura 2.1: 3-way handshake*: Troca de segmentos entre cliente e servidor para estabelecimento de conexão.*

Essa restrição pode impactar diretamente o desempenho do protocolo. Em [RCC+11], os autores argumentam que a maior parte das páginas web são organizados em pequenos objetos para *download* e que, nesses casos, a latência para carregar uma página é, em grande parte, devida ao tempo requerido para os vários estabelecimentos de conexão. Para chegar nessa conclusão, os autores fizeram análises de relatórios de servidores da Google e de estatísticas do Chrome. Na primeira análise, usaram uma amostra de bilhões de requisições para serviços da Google e mediram a latência como sendo o tempo entre o recebimento do primeiro byte da requisição e toda a resposta ser reconhecida (ter recebido ACK). O resultado foi que *handshakes* representam entre 8% e 28% da latência em requisições "frias", ou seja, aquelas que iniciam novas conexões. O custo amortizado do

handshake entre requisições "quentes" (que reaproveitam uma conexão) e "frias" foi de 5% a 7%. Na segunda análise, olhando do ponto de vista do usuário (Chrome), o *handshake* representa até 25% da latência em requisições "frias". Por esse motivo, o QUIC propõe um mecanismo novo para estabelecimento de conexão. Como será explicado em mais detalhes na seção 2.3.4, esse mecanismo permite que dados sejam enviados logo no primeiro segmento da conexão na maior parte dos casos, eliminando a necessidade de espera de 1 RTT para isso.

A ideia do QUIC de possibilitar o envio de dados sem obrigatoriamente ter que realizar passos de *handshake* não é nova. Muitos trabalhos já propuseram, de diferentes modos, a ideia de guardar alguns dados sobre uma primeira conexão no cliente e/ou no servidor para permitir que, em conexões subsequentes, clientes possam enviar dados logo no primeiro pacote da troca de mensagens. É importante ressaltar que o TCP **também permite** que dados sejam enviados no pacote SYN, porém **proíbe** que servidores entreguem dados para a aplicação antes que o estabelecimento de conexão seja concluído. Vale lembrar, também, que um dos objetivos para TCP implementar o mecanismo de *3-way handshake* é reduzir a possibilidade de conexões falsas evitando [Pos81], assim, ataques de negação de serviço. Por exemplo, um atacante poderia enviar diversas requisições GET para um servidor utilizando endereços IP forjados. Isso faria com que o servidor processasse operações potencialmente caras e enviasse respostas potencialmente pesadas para os verdadeiros computadores dos endereços IP forjados. Portanto, abordagens que permitem que o servidor repasse informações em pacotes SYN para aplicações devem procurar outras formas de se proteger desse tipo de ataque.

Por exemplo, o T/TCP, descrito na RFC 1644 [Bra94], em 1994 já propunha um mecanismo para possibilitar envio de dados em pacotes SYN. Essa variação do TCP introduz um mecanismo chamado *TCP Accelerated Open* (TAO) para contornar o *handshake* inicial. Esse mecanismo depende de um contador de conexão, o *Connection Count* (CC), que possui valores monotonicamente crescentes e que são carregados em cada segmento TCP no campo de opções. Dessa forma, um servidor T/TCP mantém um cache com o último CC que recebeu de cada cliente. Se um SYN chegar com CC maior do que o guardado, ele deve ser novo (por conta dos valores monotônicos) e pode ser entregue para a aplicação imediatamente. Caso contrário, o servidor não tem como saber se o SYN é um antigo duplicado ou recebido fora de ordem. Nessa situação, o servidor executa o *3-way handshake*. O T/TCP, porém, nunca foi amplamente implementado devido às suas graves falhas de segurança. Por exemplo, é possível que um atacante crie CCs válidos que enganarão o teste feito pelo TAO. Esses CCs válidos podem ser conseguidos por *sniffing* ou ao escolher um valor de CC grande assumindo que o valor armazenado pelo servidor não está perto do maior valor possível [DVdVKI99].

Outro exemplo é o TFO (*TCP Fast Open*) [RCC+11], uma iniciativa da Google para fornecer um mecanismo seguro que permita troca de dados em pacotes SYN. O TFO utiliza uma conexão inicial para obter um cookie de segurança e, com ele, pode enviar dados em pacotes SYN de conexões subsequentes. Para obter o cookie, o cliente indica seu interesse com uma *flag* no campo de opções do pacote SYN. O servidor gera um cookie criptografando o IP do cliente e envia o cookie no campo de opções do pacote SYNACK. O cliente, então, armazena o cookie em cache. Dessa forma, em conexões futuras o cliente envia o cookie armazenado no campo de opções do pacote SYN, junto com dados para a aplicação. Para validar o cookie, o servidor pode tanto descriptografá-lo e comparar os endereços de IP do cliente ou criptografar o endereço IP e comparar com o cookie. Se o cookie for válido, o servidor entrega os dados para a aplicação e envia um SYNACK que confirma o recebimento tanto do SYN quanto dos dados. Se o cookie não for válido, o servidor os ignora e utiliza o procedimento de *handshake* comum. Além disso, se os dados do SYN forem aceitos, o servidor pode transmitir segmentos de resposta adicionais ao cliente antes de receber o primeiro ACK. Por outro lado, se os dados do cliente não forem aceitos, eles terão que ser retransmitidos. A partir daí a conexão segue como TCP comum. O estabelecimento de conexão do QUIC, detalhado na subseção 2.3.4, foi baseado no TFO.

Estabelecimento de Conexão Segura

O protocolo TCP não especifica nenhuma função de criptografia para aumentar a segurança na transferência de dados. Com criptografia, uma conexão TCP pode melhorar serviços de segurança

como confidencialidade, integridade de dados e autenticação. Sem confidencialidade, dados sensíveis (como senhas) podem ser roubados. Sem integridade de dados, requisições podem ser alteradas para executarem outras funções ao invés da desejada. Sem autenticação, um cliente pode se conectar a um servidor forjado pensando que está se conectando com o verdadeiro. O TLS, definido na RFC 4346 [DR06], foi desenvolvido para fornecer segurança para a camada de transporte. Ele funciona criando um "canal" seguro de dados pelo qual todos os segmentos TCP serão enviados. Para a criação desse "canal" seguro de dados, o TLS depende de um *handshake* próprio. A Figura 2.2 mostra esse processo. O *handshake* do TLS começa após o estabelecimento de conexão do TCP. Primeiramente, o cliente envia uma mensagem `HELLO` para iniciar o processo. O servidor responde a essa mensagem com seu certificado para comprovar que é realmente com quem o cliente quer iniciar uma conexão. O cliente, então, cria uma chave para criptografar o resto da comunicação e a envia para o servidor. O servidor guarda essa chave e responde indicando que o *handshake* foi concluído com sucesso. A partir daí, o cliente pode começar a enviar dados para o servidor. Note que o cliente precisa esperar 2 RTTs adicionais antes de poder enviar dados para o servidor em um *handshake* TLS comum. Há alguns casos em que apenas 1 RTT adicional é necessário como, por exemplo, quando o cliente já se comunicou com o servidor previamente. Porém, para o estabelecimento de uma conexão segura é preciso que o cliente espere de 2 a 3 RTTs antes de poder enviar dados. Como veremos na subseção 2.3.4, o QUIC propõe um mecanismo para o estabelecimento de conexão segura sem que o cliente precise esperar respostas do servidor para enviar dados na maior parte delas, tendo que esperar 1 RTT caso algo dê errado.

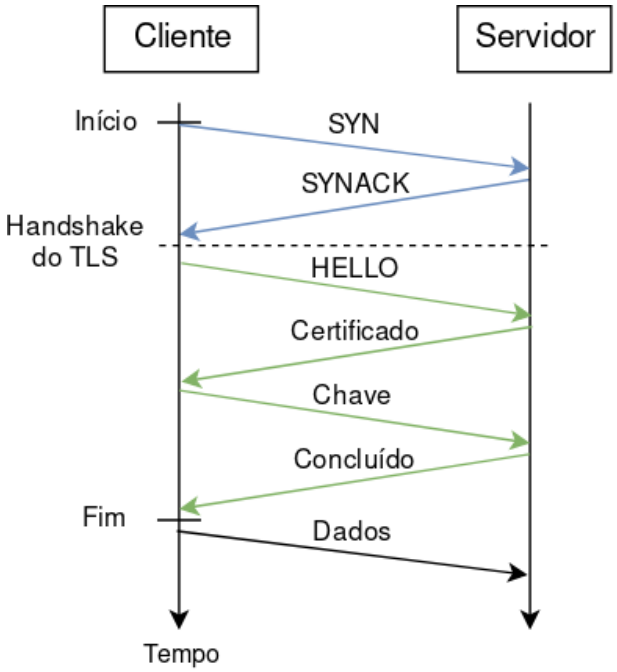

Figura 2.2: *Representação de um estabelecimento de conexão segura entre cliente e servidor.*

2.2.2 Controle de Congestionamento

O congestionamento de uma rede de computadores acontece quando muitas fontes tentam enviar dados para seus destinos a uma taxa de transmissão muito elevada [KK16]. Quando isso ocorre, os *buffers* de roteadores na rede podem encher, fazendo com que pacotes sejam descartados antes de serem processados e redirecionados. Sintomas do congestionamento da rede são o alto atraso na entrega dos pacotes e a necessidade de retransmitir pacotes por conta dos descartes, ainda que algumas vezes essas retransmissões sejam desnecessárias. Esses problemas na camada de transporte causam impacto direto nas aplicações como, por exemplo, longas demoras para o carregamento de páginas web e vídeos que "travam" constantemente enquanto estão sendo assistidos. Por isso, o controle de congestionamento é crucial para o funcionamento de redes de computadores e, em par-

ticular, da Internet. De maneira geral, há duas abordagens de controle de congestionamento: uma em que o controle é auxiliado pela rede e outra em que o controle é feito ponta-a-ponta [KK16]. Na primeira abordagem, roteadores fornecem informação explícita para remetentes e/ou destinatários sobre o congestionamento da rede. Essa informação pode ser enviada por pacotes criados pelos próprios roteadores ou por meio de uma marcação em pacotes que passam pelo roteador. Na abordagem ponta-a-ponta, a rede não oferece nenhuma sinalização explícita de congestionamento e sua presença precisa ser inferida pelo comportamento da rede como, por exemplo, pela perda de pacotes ou por atrasos.

A primeira versão do TCP não possuía controle de congestionamento. Durante a década de 1980, isso levou a eventos em que aplicações ficavam impossíveis de serem usadas porque a rede não conseguia entregar os pacotes. Esse tipo de situação ficou denominada de **colapso por congestionamento** [Nag84]. Em 1988, [Jac88] identificou esse problema e propôs diversos algoritmos que depois foram padronizados pela RFC 5861 [Not10]. Esses algoritmos são: **partida lenta** (*slow start*), **prevenção de congestionamento** (*congestion avoidance*), **retransmissão rápida** (*fast retransmit*) e **recuperação rápida** (*fast recovery*). Esses algoritmos rodam em fases diferentes de uma transmissão de dados, mas o princípio de todos eles é realizar um ajuste adequado da janela de congestionamento.

A janela de congestionamento restringe a taxa de transmissão de dados de um *host*: ele só poderá transmitir a quantidade de dados indicada por ela e só poderá enviar mais dados quando receber a confirmação de que os dados que já enviou chegaram no destino. Por exemplo, no início de uma conexão o valor dessa janela é tipicamente de 1 MSS (*Maximum Segment Size*), que é o tamanho máximo de um segmento TCP. Ou seja, no início da conexão quem envia poderá transmitir apenas 1 segmento até que um ACK para esse segmento seja recebido. Porém, a rede pode ter mais banda disponível. É o objetivo do algoritmo de partida lenta descobrir o quanto um remetente pode aumentar sua taxa de transmissão sem congestionar a rede. Para cada ACK recebido, soma-se 1 à janela de congestionamento. Na prática, esse aumento tem comportamento exponencial porque dobra a taxa de transmissão a cada RTT. Esse aumento continua até que uma perda seja identificada por *timeout*, até que a janela ultrapasse um certo valor limite, ou até que o remetente identifique uma perda por receber 3 ACKs duplicados.

No primeiro caso, o TCP altera o valor limite da janela para a metade do valor atual da janela e, em seguida, altera o valor da janela novamente para 1. Note que o limite da janela indica metade do último valor da janela para o qual foi detectado congestionamento. Por isso, quando o valor do tamanho da janela ultrapassa esse limite, o TCP sai do estado de partida lenta e entra no estado de prevenção de congestionamento.

No estado de prevenção de congestionamento, ao invés de dobrar o valor da janela, soma-se apenas 1 MSS a ela a cada RTT. Tanto o estado de partida lenta quanto o de prevenção de congestionamento se comportam da mesma maneira quando identificam perda por *timeout* ou por 3 ACKs duplicados.

Receber ACKs duplicados significa que, apesar de segmentos estarem sendo recebidos, ainda existe um mesmo segmento pendente. Caso 3 ACKs duplicados sejam recebidos, o TCP assume que um pacote foi perdido, sai do estado de partida lenta ou de prevenção de congestionamento, executa a retransmissão rápida e entra no estado de recuperação rápida. O número de 3 duplicações foi definido empiricamente, para levar em consideração pacotes que chegam fora de ordem no destino. Como a perda foi detectada por duplicação de ACKs, o TCP pode retransmitir o segmento perdido de forma imediata (retransmissão rápida) ao invés de esperar seu tempo de expiração. Pelo mesmo motivo, como a rede continua entregando pacotes, a redução na janela de congestionamento é menor do que quando ocorre perda por *timeout*: a janela é reduzida pela metade e soma-se à ela 3 MSS (para incluir os 3 ACKs duplicados recebidos) e o valor limite da janela é reduzido para a metade do valor da janela de quando os 3 ACKs duplicados foram recebidos. O TCP entra, então, no estado de recuperação rápida.

Durante a recuperação rápida, o TCP adiciona 1 MSS à janela para cada ACK duplicado recebido por conta do segmento ausente que o fez entrar nesse estado. Quando o ACK para o

segmento ausente chegar, o TCP força o valor da janela de congestionamento a ser igual ao valor limite da janela e entra em estado de prevenção de congestionamento. Se ocorrer um *timeout*, o comportamento é igual ao que acontece no estado de partida lenta.

A Figura 2.3 mostra um exemplo de como a janela de congestionamento é ajustada pelo controle de congestionamento do TCP ao longo de uma transmissão de dados. Nesse exemplo, o algoritmo de partida lenta aumenta o valor da janela exponencialmente até ultrapassar seu limite inicial, acionando a prevenção de congestionamento. Ao receber 3 ACKs duplicados, o TCP aciona os algoritmos de retransmissão e recuperação rápidas. Finalmente, quando ocorre o *timeout* de algum segmento, o TCP volta para o estado de partida lenta. Note que o comportamento do controle de congestionamento do TCP, depois de entrar pela primeira vez no estado de prevenção de congestionamento e até ocorrer o *timeout* consiste em acréscimos lineares e decréscimos multiplicativos na janela (AIMD, *Additive-Increase Multiplicative-Decrease*), formando o comportamento de "serrote" que caracteriza a constante busca do TCP por banda disponível.

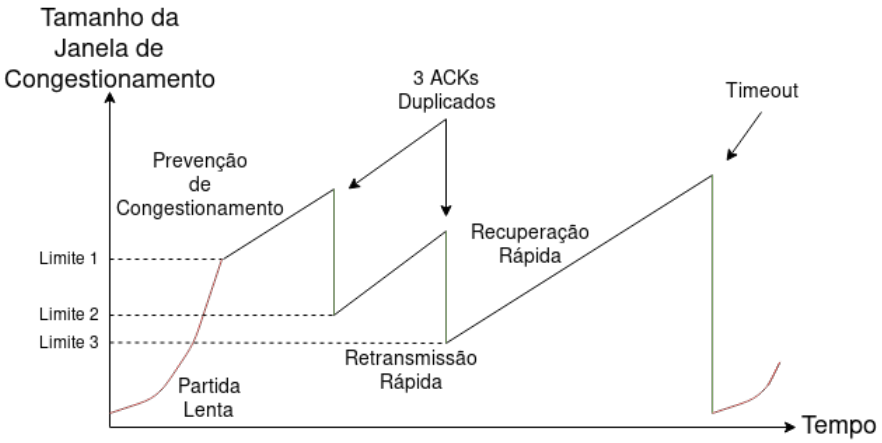

Figura 2.3: *Exemplo de variação do tamanho da janela de congestionamento no TCP com o passar do tempo.*

Contudo, o controle de congestionamento do TCP apresenta diversos problemas. Dentre eles estão a incapacidade de diferenciar motivos de perda de pacotes [TM02], o favorecimento de fluxos com RTTs pequenos [LM97] e a subutilização da largura de banda [WPY+04]. Muita pesquisa tem sido feita com o objetivo de melhorar o controle de congestionamento ponta-a-ponta do TCP [ATRK10]. No contexto deste trabalho, porém, duas propostas de controle de congestionamento são de especial importância: o CUBIC e o BBR.

O CUBIC [HRX08] é, na verdade, uma versão melhorada do BIC-TCP [XHR04]. O BIC-TCP era o algoritmo de controle de congestionamento padrão do Linux, a partir da versão 2.6.8. Com o lançamento do CUBIC, ainda em 2006 o Linux substituiu o BIC-TCP por ele. Desde a versão 2.6.18 até a versão atual, a 4.6.3, o CUBIC permanece como o algoritmo de controle de congestionamento padrão do Linux e uma versão modificada é usada como algoritmo padrão do QUIC. O CUBIC, assim como o TCP, detecta congestionamento por perda de pacotes. Esse algoritmo utiliza uma função cúbica para ajustar a janela de congestionamento em termos do tempo passado desde a última perda de pacote. Essa variação mantém uma variável adicional: o tamanho máximo da janela de congestionamento. Esse valor indica o tamanho da janela antes da detecção de uma perda de pacote. Quando há uma perda, a janela é reduzida por uma fator multiplicativo constante e o tamanho máximo da janela é atualizado. A partir daí, o CUBIC começa a incrementar a janela utilizando a parte côncava da função cúbica. A função é ajustada para que seu *plateau* seja igual ao tamanho máximo da janela. Quando passar do tamanho máximo, o crescimento da função é feito por sua parte convexa. Além disso, o CUBIC possui um "modo TCP". Esse modo é ativado quando o valor de ajuste do tamanho da janela que o CUBIC calcula é menor do que o tamanho de janela que o TCP alcançaria depois da última perda. Nesses casos, a janela que o TCP alcançaria é usada. O CUBIC também tem uma funcionalidade opcional de convergência rápida (*fast convergence*)

para ajudar fluxos novos na rede. Se estiver habilitada, fluxos já existentes reduzirão sua janela um pouco mais em comparação com o modo comum. Com essa heurística, os autores acreditam que um novo fluxo terá mais tempo para conseguir maior taxa de transmissão, ou seja, para aumentar o tamanho de sua janela de congestionamento. O CUBIC utiliza melhor os recursos da rede, não prioriza fluxos com RTTs de diferentes tamanhos e é justo com outros fluxos TCP.

Já o BBR (*Bottleneck Bandwidth and Round-trip propagation time*) [CCG+16] é um algoritmo de controle de congestionamento publicado em 2016, com implementação no Linux e já utilizado pela plataforma de nuvem da Google [BBR]. Diferentemente do TCP e do CUBIC, o BBR não identifica congestionamento baseando-se em perdas de pacote. Ao invés disso, o BBR faz estimativas melhores de largura de banda e RTT para identificar que um caminho está ficando congestionado e evitar que roteadores descartem pacotes por estarem com seus *buffers* cheios. Para conseguir estimativas melhores, o BBR usa um histórico recente dos RTTs e da quantidade de dados que o destino sinalizou que recebeu para estimar o tempo de propagação e a largura de banda no gargalo do caminho da transmissão. Portanto, toda vez que um ACK é recebido essas estimativas são atualizadas. Outro aspecto característico do BBR é o fato de ele dar intervalos de espera entre o envio de pacotes (veja a seção 2.3.4) para que a taxa de chegada de pacotes no enlace gargalo seja igual à taxa de saída. Para capturar aumentos na banda disponível, de tempos em tempos o algoritmo passa 1 RTT mínimo estimado com um intervalo menor de espera entre o envio de pacotes, para aumentar a taxa de envio. Se a banda disponível no gargalo não aumentou, uma fila será criada nele, aumentando o RTT. Essa fila é removida no próximo RTT mínimo estimado aumentando o intervalo de espera entre o envio de pacotes. Se a banda disponível aumentou, a estimativa da banda disponível no gargalo é atualizada, junto com o intervalo de espera para transmissão de pacotes. Em uma transmissão usando o BBR, em seu início o algoritmo dobra a taxa de envio enquanto a taxa de entrega de pacotes crescer (estado de *Startup*). Quando a largura de banda gargalo é encontrada, o BBR usa o inverso da taxa de aumento usada na fase de *Startup* para drenar a fila formada no gargalo (estado de *Drain*). Finalmente, o algoritmo passa para a fase de sondagem (*ProveBW*) para capturar aumentos na banda disponível, como explicado acima. Quando uma atualização da estimativa para o tempo mínimo de propagação não é realizada por algum tempo, o BBR reduz a quantidade de pacotes em transmissão por pelo menos 1 RTT. Dessa forma, a fila no gargalo diminui e quem envia poderá perceber RTTs menores. Essa sondagem pelo tempo mínimo de propagação é o estado de *ProbeRTT* e é útil para fluxos novos na rede. Isso porque eles podem superestimar o tempo de propagação mínimo caso comecem sua transmissão quando outros fluxos temporariamente criaram uma fila. Experimentos mostraram que o BBR tem desempenho melhor que o CUBIC [CCG+16] e, por isso, a Google está trabalhando para incorporá-lo no código do QUIC.

2.3 QUIC

O QUIC é um protocolo de transporte que roda acima do UDP. Ele fornece multiplexação e controle de fluxo equivalentes aos do HTTP/2 (*Hypertext Transfer Protocol*), segurança equivalente ao do TLS (*Transport Layer Security*) e controle de congestionamento semelhante ao do TCP. O desenvolvimento e especificação do QUIC começaram no início de 2012 pela Google e desde então o protocolo vem sendo evoluído e implantado nos servidores da Google e no navegador *Chromium*. O QUIC foi projetado para atender às novas funcionalidades vindas do HTTP/2, assim como absorver a camada de segurança. Os próprios desenvolvedores da Google dizem que *"na superfície, o QUIC é muito similar a TCP+TLS+HTTP/2 implementado usando UDP"* [QUI]. A Figura 2.4 mostra a posição na qual o QUIC se encaixa no modelo de redes atual: abaixo da camada de aplicação, incorporando a camada de segurança e acima do UDP. Incorporar a camada de segurança foi necessário porque o QUIC usa técnicas de estabelecimento de conexão que não estão previstas no TLS 1.2. Porém, essa parte do protocolo será substituída pelo TLS 1.3 no futuro, quando a especificação dele estiver pronta. No momento desta escrita, o TLS 1.3 ainda é um *draft* da IETF, assim como o QUIC. Note também que, ao usar o QUIC, o HTTP/2 é desacoplado da camada de transporte. O acoplamento existe com o TCP porque o HTTP/2 implementa os

conceitos de fluxos de dados e multiplexação. Um fluxo de dados, no HTTP/2, é uma sequência de blocos de dados independentes e bidirecionais, identificados por um número inteiro e trocados entre o cliente e o servidor em uma conexão. Dessa forma, o HTTP/2 permite que uma única conexão possa conter múltiplos fluxos abertos concorrentes, com cada ponta da conexão intercalando blocos de dados de múltiplos fluxos. A divisão de uma transferência de dados em diversos fluxos lógicos de dados é chamada de multiplexação. O TCP, entretanto, possui um único fluxo de dados em cada conexão. Dessa forma, o HTTP/2 precisa duplicar mecanismos de controle de fluxo, já implementados pelo TCP, para que fluxos na mesma conexão não interfiram um no outro. Para isso, o HTTP/2 precisa ler informações específicas diretamente de *buffers* do TCP. O uso do QUIC, que também multiplexa conexões, possibilita que esse controle seja feito apenas no protocolo de transporte. Consequentemente, o controle de fluxo poderá ser removido do HTTP/2 e este poderá fornecer apenas uma API (*Application Programming Interface*) para comunicação com o QUIC.

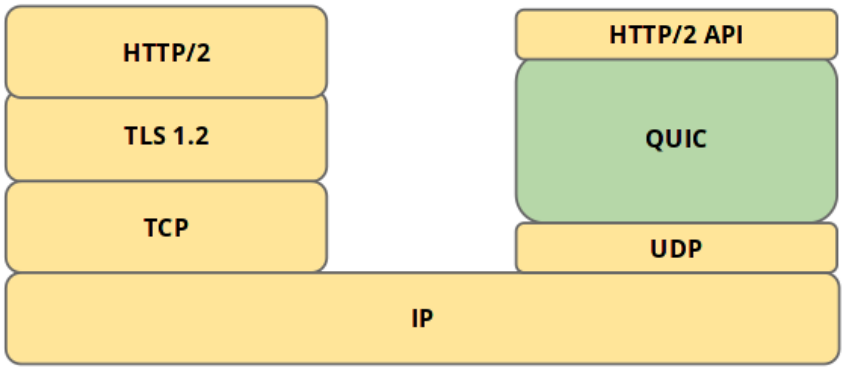

Figura 2.4: *Posição do QUIC no modelo de redes atual. [Iye].*

Toda especificação que há sobre o QUIC está escrita em documentos efêmeros: documentos da Google Docs ou *drafts* de RFCs (*Request For Comments*). Nesta seção, a maior parte dos conceitos baseia-se no documento escrito por Jim Roskind [Ros], pois nele estão contidos os princípios de funcionamento e decisões de projeto que devem mudar pouco ao longo do tempo.

2.3.1 Motivações

A maior motivação para o desenvolvimento do QUIC foi o mau "encaixe" de algumas funcionalidades vitais do HTTP/2 com o TCP. Uma delas, como já foi dito, é a capacidade de fazer multiplexação por meio de fluxos de dados. Essa funcionalidade foi implementada para resolver um problema comum do HTTP/1.1: o *head-of-line blocking*. Nessa versão do protocolo, foi introduzida a função de *pipelining*. Antes disso, um cliente era obrigado a esperar a resposta de um pacote antes que pudesse enviar o próximo, mesmo que nesse tempo ele ficasse ocioso. A intenção do *pipelining* era habilitar um cliente a enviar múltiplas requisições de uma vez, sem que precisasse esperar a resposta da primeira para enviar a próxima. Esse mecanismo também possibilitava o processamento paralelo de requisições no servidor. A Figura 2.5 mostra uma comparação entre requisições sem e com *pipelining*. Com ela podemos ver o ganho em termos de redução da latência que, teoricamente, esse mecanismo consegue: com o *pipelining*, todos os dados seriam transmitidos em um intervalo de tempo menor do que no ambiente sem *pipelining*.

Contudo, o *pipelining* possui uma característica que, no final das contas, faz com que a comunicação continue lenta em situações recorrentes. Isso ocorre porque, para que o cliente consiga determinar correspondência entre requisição e resposta, o servidor precisa enviar as respostas na ordem em que as requisições chegaram. A imagem da esquerda da Figura 2.6 ilustra um caso em que essa situação aparece. Suponha que o cliente envie múltiplas requisições para o servidor, representadas pelas setas 1, 2 e 3. O servidor as processa em paralelo. O pacote representado pela seta 1 (para facilidade da explicação vamos considerar que 1 pacote equivale a 1 requisição) é processado

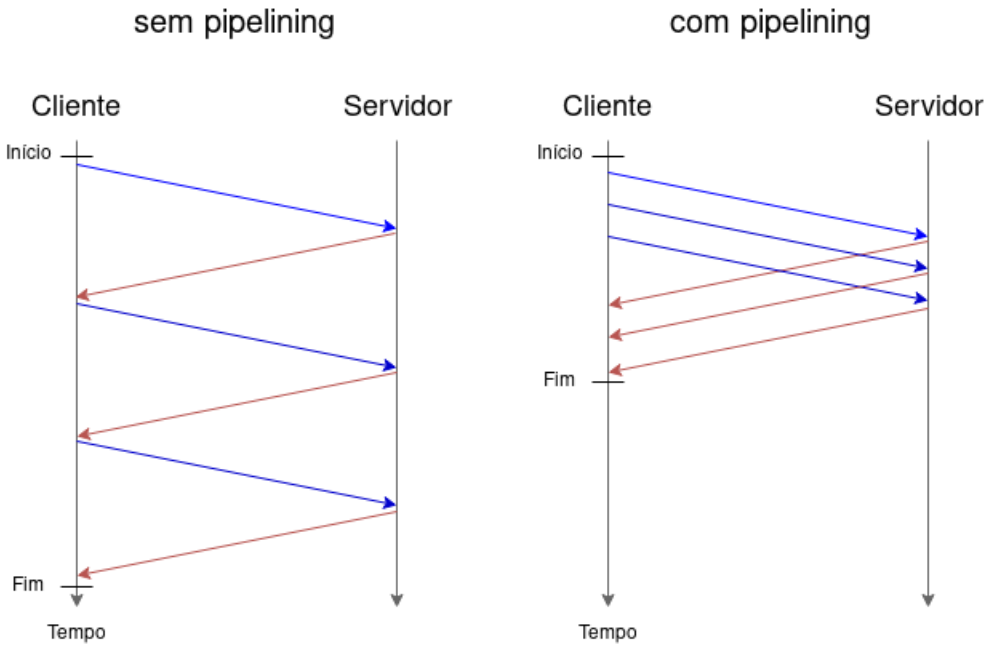

Figura 2.5: *Comparação de requisições sem (esquerda) e com (direita)* pipelining. *Figura adaptada de [Pip].*

rapidamente e é imediatamente respondido. Já o pacote representado pela seta 3 demora mais para ser processado, mas sua resposta fica pronta antes do pacote da seta 2 terminar seu processamento. Por exemplo, o pacote da seta 2 pode envolver alguma transação complexa em um banco de dados. Nesse caso, apesar de o pacote da seta 3 já estar pronto para ser enviado, ele terá que esperar até que o pacote da seta 2 tenha sido processado e sua resposta enviada. Esse problema, em que um fluxo de dados fica impedido de continuar por conta de um pacote que demora a ser processado ou é perdido, é chamado de *head-of-line blocking*.

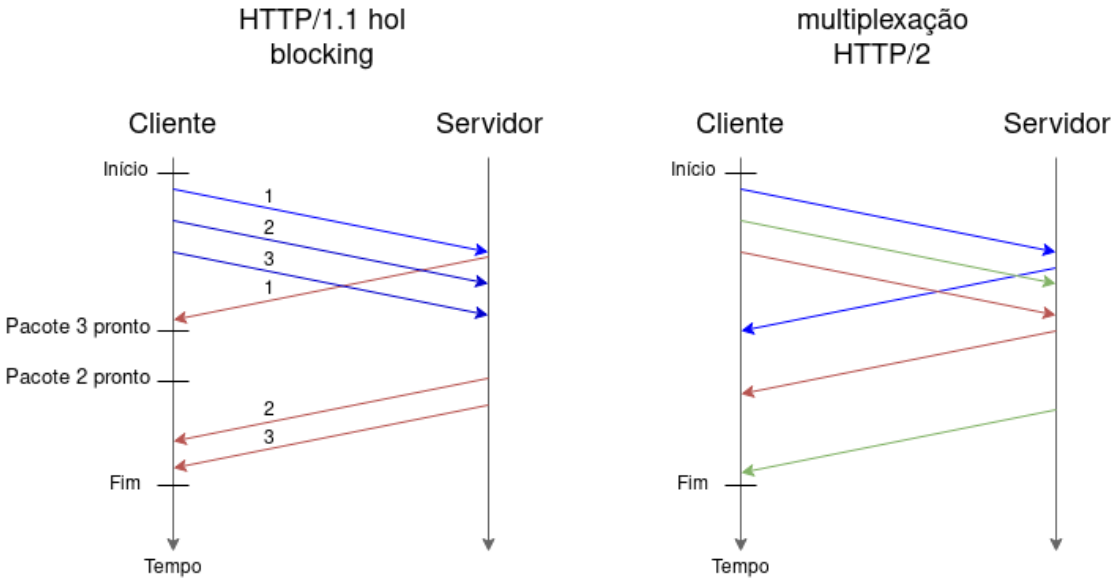

Figura 2.6: *Na esquerda, uma situação de head-of-line blocking no HTTP/1.1. Na direita, a multiplexação de pacotes no HTTP/2. Cada fluxo de dados é representado por uma cor diferente.*

A multiplexação do HTTP/2 mitiga esse problema. Como podemos ver na imagem da direita da Figura 2.6, agora, cada requisição pode ser feita em um fluxo de dados separado. No recebimento dos pacotes pelo servidor, cada pacote continua podendo ser processado em paralelo porém, ao término do processamento, o servidor já pode enviar a resposta. Isso porque a correspondência

entre requisição e resposta é feita pelo próprio fluxo. Portanto, o servidor não precisa esperar para enviar pacotes que recebeu em fluxos diferentes. Quando há *head-of-line blocking*, o efeito só é sentido pelo fluxo do pacote que causou o problema, não prejudicando a comunicação no geral. Por exemplo, se os pacotes do fluxo verde estiverem demorando muito tempo para serem processados, apenas seu fluxo correspondente será afetado. Os pacotes dos fluxos vermelho e azul continuam a transferência normalmente.

Apesar de o problema de *head-of-line blocking* estar amenizado na camada de aplicação, a forma que o TCP funciona faz com que esse benefício não seja tão bem percebido. A Figura 2.7 ilustra uma situação em que esse problema pode ocorrer com o TCP, na qual a perda do pacote vermelho faz com que todos os próximos pacotes que chegam sejam obrigados a esperar antes de serem repassados para a aplicação, ainda que sejam de fluxos diferentes. Essa situação está associada com os mecanismos de garantia de entrega e entrega ordenada dos pacotes empregados pelo TCP. Ainda que a aplicação crie múltiplos fluxos para enviar as requisições, eles passam pelo TCP como se fossem um único fluxo. Quando há uma perda de pacote, o mecanismo de entrega ordenada impede que o TCP entregue os pacotes que já chegaram para a aplicação. Ao invés disso, esses pacotes terão que esperar que o pacote pendente seja retransmitido, configurando um cenário de *head-of-line blocking*.

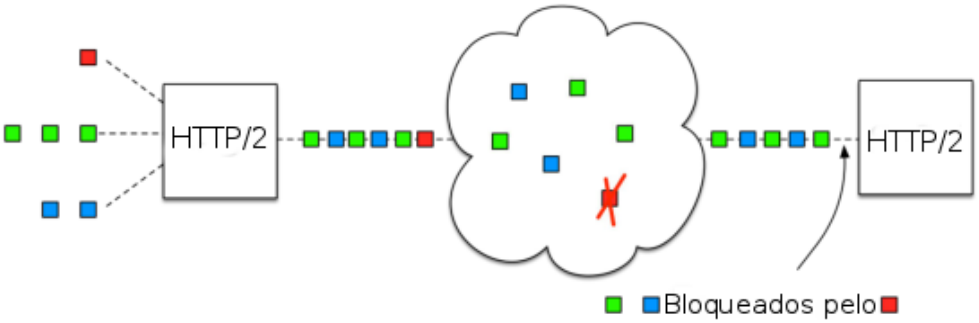

Figura 2.7: *Head-of-line blocking no TCP. Figura adaptada de [Sha].*

Existe outro problema relacionado com o TCP, mas agora com seu controle de congestionamento. Uma única conexão HTTP/2 é usada para substituir K conexões separadas (não multiplexadas). Quando um pacote de uma conexão HTTP/2 é perdido, a janela de congestionamento é reduzida drasticamente para evitar congestionar a rede. Por outro lado, a perda de um pacote dentre K conexões TCP não multiplexadas causa essa redução na janela de congestionamento em apenas uma das conexões. Dessa forma, o controle de congestionamento do TCP favorece múltiplas conexões ao invés de uma conexão multiplexada. Essa situação é mais comum do que parece: navegadores abrem múltiplas conexões TCP para fazer o *download* concorrente de recursos de uma página web, quando esses recursos estão espalhados por diferentes servidores na Internet, com o objetivo de reduzir o tempo total de carregamento da página.

Por fim, todas as conexões QUIC são criptografadas, porém aspectos de funcionamento do TLS atrasam a comunicação de dados. Um desses aspectos é a retomada de sessão. Devido à implementação e não a requisitos de segurança, o *handshake* do TLS precisa de, pelo menos, 1 RTT adicional antes que dados possam ser transmitidos. Outro aspecto é uma dependência para descriptografar mensagens. Em sua versão 1.0, o TLS requeria que pacotes fossem descriptografados em ordem. Na versão 1.1 do TLS e no DTLS (*Datagram Transport Layer Security* - uma derivação do TLS para segurança no UDP), esse problema foi resolvido incluindo vetores de inicialização, com o custo de aumentar o tamanho do pacote. O QUIC procura resolver esses problemas, implementando o próprio mecanismo de segurança para acelerar o estabelecimento de conexão e minimizar o tamanho das mensagens. Esse mecanismo futuramente será substituído pela versão 1.3 do TLS, que contará com as mesmas funcionalidades de segurança implementadas pelo QUIC.

2.3.2 Principais Objetivos

Os problemas apresentados na seção anterior motivaram o desenvolvimento do QUIC tendo como principais objetivos:

1. **Reduzir o *head-of-line blocking*.** Principalmente em casos de perda de pacotes;
2. **Reduzir a latência.** Tanto no estabelecimento de uma conexão segura quanto em situações em que ocorrem perda de pacotes;
3. **Melhorar o controle de congestionamento.** O algoritmo deve levar em consideração a multiplexação de fluxos, ter desempenho comparável e ser amigável ao TCP. Além disso, o controle de fluxo deve ser específico para cada fluxo e,
4. **Segurança.** Possuir garantias de segurança comparáveis ao TLS, sem precisar descriptografar pacotes em ordem.

2.3.3 Implicações de Usar UDP

Evoluir o TCP é plausível mas não viável. Isso porque a implantação depende de atualizações nos sistemas operacionais e modificações mais significativas possivelmente fariam com que o tráfego fosse bloqueado por *middleboxes*. Por isso a decisão de construir o QUIC acima do UDP. O UDP é um protocolo da camada de transporte que não oferece as principais funcionalidades implementadas no TCP. Por exemplo, o UDP é um protocolo que não precisa de conexão, não implementa controle de congestionamento, não garante entrega e não impede que segmentos sejam recebidos fora de ordem. Portanto, se uma aplicação quiser qualquer um desses benefícios, precisará implementá-los. Em um primeiro momento, a impressão que temos é que usar o UDP complica o processo de desenvolvimento. Porém, a maior vantagem é que conseguimos liberdade para experimentar técnicas novas para cada uma dessas funcionalidades. Soma-se a isso o fato de que, como essas alterações serão feitas em uma aplicação (que fica em nível de usuário), a implantação das técnicas se torna mais rápida. Agora, só é preciso que os clientes e servidores atualizem suas aplicações (e não o sistema todo), não dependendo de atualizações de *middleboxes*. Esse cenário possibilita testes e medições reais de desempenho das novas técnicas empregadas, gerando *feedback* mais rápido sobre o que funciona ou não. Pode-se inclusive incluir essas informações no próprio ciclo de desenvolvimento do protocolo, promovendo entregas contínuas e melhorando o produto final. Por exemplo, recentemente foi provado que era possível que um atacante conseguisse se passar por um servidor QUIC real indefinidamente [ASS+16]. Por conta das iterações de implantação mais curtas, os desenvolvedores do QUIC puderam lançar imediatamente uma correção temporária. Menos de um mês depois, na versão seguinte, lançaram uma correção definitiva.

UDP e NATs

Apesar de ter muitas vantagens para experimentação e implantação rápidas, implementar o QUIC em cima do UDP possui grandes dificuldades: como fazer o UDP ser baseado em conexões e como superar servidores NAT (*Network Address Translation*), pois esses podem impossibilitar a real utilização do protocolo.

Para exemplificar como servidores NAT podem dificultar conexões usando UDP, vamos comparar o funcionamento de um deles ao fechar uma porta quando há conexões TCP e quando há tráfego UDP. No caso do TCP, o servidor NAT pode enviar um pacote RST (*Reset* - que indica encerramento de conexão) para as duas pontas como forma de notificação. Já no caso do UDP, como é esperado não haver conexões, não há notificação. Uma vez que o servidor NAT não está mais escutando na porta que estava sendo usada pelo UDP, o servidor de dados fica sem meios de enviar dados para o cliente. O que é pior: o tráfego do servidor de dados é simplesmente perdido, sem notificações do que está ocorrendo. Como uma complicação adicional, se o cliente tenta enviar mais dados, o servidor NAT pode começar a escutar em uma nova porta. Essa nova porta pode

possuir o campo de porta origem diferente para o servidor de dados, mesmo que o QUIC tenha que ver o tráfego como uma continuação da conexão em vigor.

Para superar o problema de continuidade da conexão, o QUIC não define uma conexão como um par contendo IP e porta de origem. Ao invés disso ele usa um CID (*Connection Identifier*) que é válido durante toda a conexão. O CID é um número pseudo-aleatório e único entre as conexões aceitas pelo servidor. Ele é tipicamente proposto no primeiro pacote enviado para o servidor pelo cliente e estará presente em todos os pacotes trocados na conexão criada.

Para superar o problema de falta de notificação quando um servidor NAT deixa de escutar em uma porta, o QUIC propõe que o cliente envie pacotes de *keep-alive* periodicamente. Como esse tráfego não está efetivamente transmitindo dados "reais", é desejável que a frequência de envio seja mínima, especialmente em dispositivos móveis por conta do consumo de energia. A ideia é enviar um *keep-alive* pouco antes do tempo de expiração da tabela de entrada do servidor NAT ou a um passo constante.

2.3.4 Funcionamento

Estabelecimento de Conexão

O QUIC combina os *handshakes* de segurança e da camada de transporte, com o objetivo de reduzir o número de RTTs para estabelecer uma conexão segura [Lan]. Na maior parte das conexões, o cliente pode enviar dados imediatamente para o servidor sem esperar uma resposta. Essa situação possui ganhos de desempenho significativos em relação à conexão comum de TCP com TLS, que pode levar de 1 a 3 RTTs para ser iniciada. Esses ganhos são mais perceptíveis em transações mais curtas, nas quais o estabelecimento de conexão é responsável pela maior parcela do tempo total de transmissão.

No TLS, o servidor escolhe parâmetros de segurança para cada conexão, baseado no que o cliente anuncia que dá suporte. Já no QUIC, as preferências do servidor são enumeradas e estáticas. Essas preferências constituem o "*server config*", que possui um tempo de expiração e é assinado pela chave privada do servidor. Como ele é estático, uma única assinatura é válida para várias conexões. Dessa forma, o cliente pode usar essas informações estáticas do servidor para tentar estabelecer uma conexão e imediatamente enviar dados.

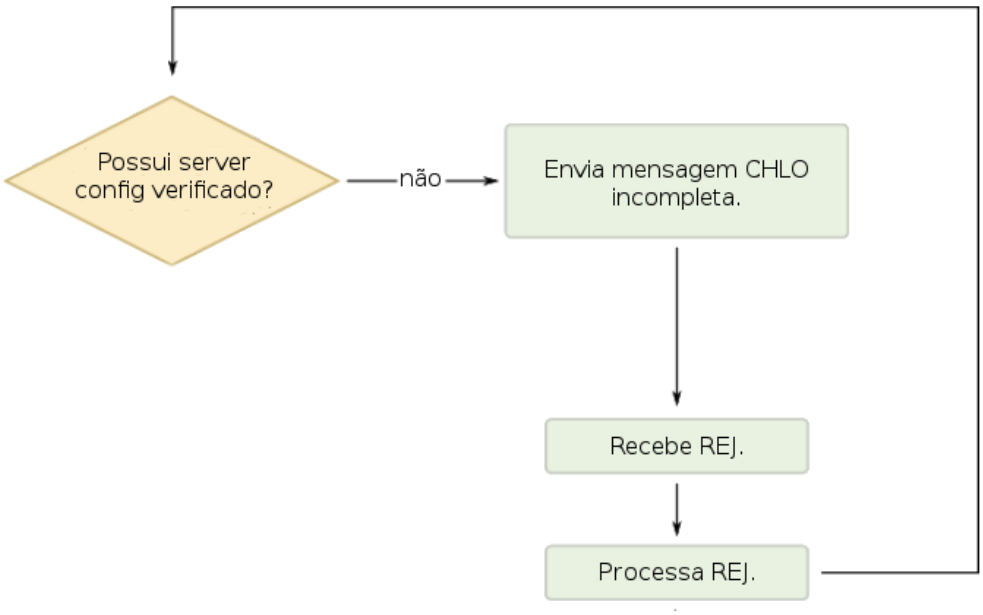

Figura 2.8: *Passos do* ***handshake*** *quando o cliente não possui* ***server config*** *autenticado e verificado. Adaptado de [Lan].*

A Figura 2.8 mostra a situação em que o cliente ainda não possui *server config* verificado.

Como o cliente não possui informação alguma sobre o servidor, ele precisará enviar uma mensagem CHLO (*Client Hello*) incompleta antes de tentar estabelecer a conexão com o servidor. O objetivo é conseguir o *server config* e provas de autenticidade. Como resposta a uma mensagem CHLO incompleta, o servidor deve enviar uma mensagem REJ (*Rejection*). Ela conterá informação que o cliente pode usar para um próxima tentativa de *handshake*. Por exemplo, uma mensagem REJ pode conter as provas de autenticidade e o *server config* que o cliente precisa. Nessa situação, o estabelecimento de conexão necessitará de pelo menos 1 RTT antes que o cliente possa fazer uma requisição ao servidor.

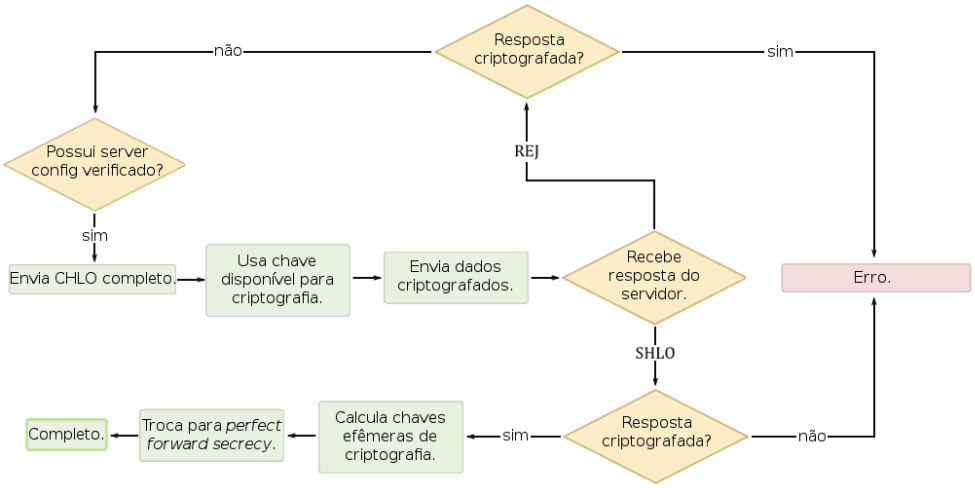

Figura 2.9: *Passos do* ***handshake*** *quando o cliente já possui* ***server config*** *autenticado e verificado. Adaptado de [Lan].*

A Figura 2.9 mostra a situação em que o cliente já possui *server config* autenticado e verificado. Com isso, ele pode enviar uma mensagem CHLO completa, ou seja, com o *server config*. O cliente, inclusive, pode começar a enviar dados. Todos os dados devem ser criptografados com a chave pública do servidor. O servidor aceitará ou rejeitará o *handshake*. No caso de rejeição, o servidor enviará uma mensagem REJ. Todos os dados enviados pelo cliente, criptografados com a chave que ele tinha, devem ser considerados perdidos. Eles devem ser retransmitidos usando novas chaves provenientes de uma nova tentativa de estabelecer conexão. No caso em que a conexão é estabelecida com sucesso, o servidor envia uma mensagem SHLO (*Server Hello*). Nessa situação, o estabelecimento de conexão ocorre de forma imediata e os dados enviados pelo cliente podem ser processados e respondidos normalmente pelo servidor, sem a necessidade de mais trocas de dados para autenticação e criação de uma comunicação segura. Essa situação é chamada de estabelecimento de conexão em 0-RTT e, quando ela acontece, o ganho de desempenho em relação ao estabelecimento de conexão segura com TCP e TLS 1.2 é significativo, principalmente para transferências de poucos dados. Na mensagem SHLO, o servidor também envia um valor **efêmero** que será usado por ele e pelo cliente para calcular novas chaves de criptografia, específicas dessa sessão. Os próximos dados deverão ser criptografados com essa chave efêmera. Com isso, o QUIC tem como característica a *perfect forward secrecy* [Kau05]. Isso significa que sessões passadas estarão protegidas se futuras chaves secretas ou senhas forem comprometidas. Em outras palavras, se alguém armazenou todo o histórico de dados transferidos por um computador, a descoberta de uma chave secreta no futuro não o habilitará a descriptografar as mensagens trocadas no passado. Quando a conexão começa a usar *perfect forward secrecy* o estabelecimento de conexão está completo.

Multiplexação, Transferência de Dados e Controle de Fluxo

Uma conexão que usa QUIC mantém um único espaço de número de sequência para controle de congestionamento e recuperação de falhas. Esse número é compartilhado pelos fluxos da conexão. Todos os dados são enviados dentro de fluxos, incluindo os *handshakes*, mas os ACKs são para os pacotes QUIC. Os ACKs, no QUIC, apontam o maior número de sequência recebido, seguido por uma lista de intervalos de números de sequência dos pacotes pendentes.

Fluxos no QUIC são modelados de forma semelhante aos fluxos HTTP/2. Eles podem ser criados tanto pelo cliente quanto pelo servidor, podem mandar dados concorrentemente com outros fluxos e podem ser cancelados. Os blocos de dados são processados na ordem em que eles são recebidos. Com a transferência de pacotes UDP em fluxos, o QUIC mitiga o problema de *head-of-line blocking* presente no TCP. Portanto, mesmo que um pacote de um fluxo seja perdido, pacotes de outros fluxos continuarão podendo ser recebidos, processados e repassados para a aplicação.

No QUIC, o controle de fluxo é implementado tanto para um fluxo individual quanto para a conexão como um todo. No caso de fluxos individuais, o destinatário indica a quantidade máxima de dados que quer receber em cada fluxo. Conforme eles chegam em um determinado fluxo, o destinatário avisa o remetente que aumentou o limite aceito naquele fluxo, permitindo que o remetente envie mais dados. Já para o caso da conexão como um todo, o controle de fluxo do QUIC limita o *buffer* agregado que um destino quer alocar para a conexão. É igual ao controle de fluxo para fluxos individuais, mas a indicação da quantidade máxima de dados que será aceita e os próprios dados recebidos são a soma de todos os fluxos ativos. O protocolo também implementa ajuste automático na quantidade máxima de dados aceita, tanto para fluxos quanto para a conexão. Ou seja, o QUIC aumenta o limite que um destinatário pode receber se há indicativo que a taxa de transmissão do remetente está sendo restringida e diminui o limite se o destinatário está demorando para processar os dados que está recebendo.

Cabeçalhos e Dados Autenticados e Criptografados

Os pacotes QUIC sempre são autenticados e a maior parte dos dados são criptografados. A razão para isso é evitar injeção de dados e manipulação de cabeçalhos nos pacotes por terceiros. Esse problema aparece no TCP, pois seu cabeçalho é enviado sem criptografia e não é autenticado. Manipulações comuns são a alteração da janela de recebimento e sobrescrita do número de sequência. Algumas dessas manipulações podem ser ataques, mas também podem ser mecanismos usados por *middleboxes* na tentativa de melhorar o desempenho do TCP. Apesar de bem intencionadas, na prática, essas mudanças limitam não apenas a evolução dos protocolos, mas também a viabilidade de implantação. Um exemplo disso é o MPTCP (*Multipath TCP*). As modificações feitas por *middleboxes* quebram o funcionamento desse protocolo. A solução encontrada foi incluir um campo de *checksum* para detectar se houve mudanças. Em caso positivo, o MPTCP simplesmente termina todos os subfluxos afetados pela modificação e em último caso passa a funcionar como um TCP comum [RPB+12] [mpt]. Portanto, no QUIC, até mesmo as partes do cabeçalho do pacote que não são encriptadas são autenticadas pelo recebedor. Dessa forma, o QUIC protege conexões desse tipo de influência externa.

Forward Error Correction

Para recuperar um pacote perdido sem esperar que esse pacote seja detectado e então retransmitido, o protocolo QUIC implementa um esquema de correção de erros, o FEC (*Forward Error Correction*) baseado em XOR (*Exclusive Or*) [Swe]. Um pacote FEC contém a paridade dos pacotes no grupo FEC. Se um pacote do grupo é perdido, o conteúdo desse pacote pode ser recuperado usando o pacote FEC e os outros pacotes do grupo. O remetente pode decidir se vai enviar pacotes FEC para otimizar certas situações como o início e o fim de uma requisição e para pacotes de cabeçalho comprimido (para evitar *head-of-line blocking*). A Figura 2.10 ilustra um exemplo do uso de um pacote FEC. Nesse caso, o grupo FEC é composto pelos pacotes 1, 2 e 3. O pacote FEC é formado pela função XOR desses três pacotes. Se um deles for perdido no meio do caminho, como

o pacote 2 da figura, e todos os outros pacotes chegarem no destino, o servidor pode usar o pacote FEC para calcular a sequência de bits que formava o pacote 2. Essa ação do servidor evita a espera para identificar que um pacote foi perdido, bem como sua retransmissão.

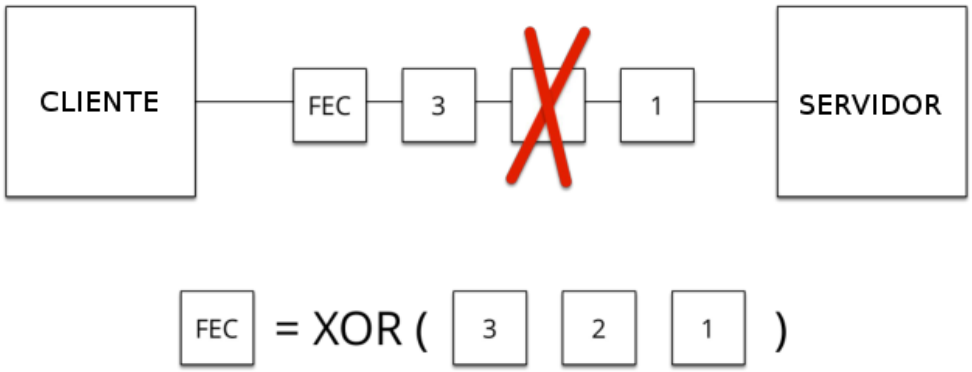

Figura 2.10: *Esquema de forward error correction, baseado em XOR, do QUIC. [Sha].*

Como os pacotes FEC aumentam o congestionamento da rede, eles são tratados da mesma forma que pacotes comuns, do ponto de vista do controle de congestionamento. Ou seja, se um pacote FEC é enviado ele "gasta" espaço da janela de congestionamento (que poderia ser usado para pacotes comuns). Da mesma forma, se um pacote FEC é perdido ele causa redução na janela de congestionamento. Apesar disso, a janela de congestionamento não bloqueia o envio de um pacote desse tipo.

A principais vantagens que uma abordagem de correção de erros baseada em XOR possui são o baixo custo computacional e a facilidade de implementação. Além disso, os pacotes originais não precisam mudar. Portanto, essa abordagem permite que os dados sejam processados incrementalmente, ao contrário de outros mecanismos de correção de erros que requerem que múltiplos pacotes cheguem antes que possam ser processados. Em contrapartida, essa abordagem baseada em XOR pode recuperar apenas um único pacote. Se dois ou mais pacotes do grupo são perdidos, o pacote FEC é desperdiçado e todos os pacotes perdidos precisam ser retransmitidos. Evidentemente, o desperdício desse pacote também acontece quando nenhum pacote do grupo é perdido.

A Google fez diversos experimentos em cenários diferentes para avaliar a eficiência dessa abordagem. As métricas calculadas foram: o tempo antes que um vídeo comece a ser executado (*join latency*); a porcentagem do tempo de execução que um vídeo fica "travado" (*rebuffer rate*); o tempo gasto para carregar os resultados de uma busca (*search latency*) e, o tempo que uma requisição demorou do ponto de vista do Chrome (*chrome latency*). Curiosamente, os resultados indicaram que a primeira versão do mecanismo de FEC do QUIC tinha efeitos mínimos na latência e **piorava** as métricas de QoE (*Quality of Experience*) para vídeos no YouTube. Tudo isso a um custo de transmitir mais dados e aumentar a complexidade do código. Outros experimentos também comprovam que o desempenho do QUIC cai ao utilizar esse esquema de FEC [CDCM15]. Por conta disso, a Google decidiu parar de usar o FEC com abordagem baseada em XOR. Eles argumentam que para aplicações com a característica de perder pacotes no final das transações, o TLP (*Tail Loss Probe*) pode ser mais eficiente e mais simples (só quem envia os dados precisa ser modificado). Esse algoritmo envia segmentos de sondagem para ativar ACKs duplicados com a intenção de invocar o estado de *fast recovery* mais rapidamente que um *timeout* [DCCM13]. O mecanismo de FEC é mais um caso que mostra o valor de se poder implantar técnicas novas em produção rapidamente: apesar de parecer uma técnica interessante, ao ser testada em ambientes reais os resultados não foram os esperados. Ao detectar uma falha ou queda de desempenho, também é rápido para implantar uma nova versão do protocolo sem, ou com alterações de, uma determinada técnica.

Packet Pacing

No TCP, os pacotes são enviados ao máximo da taxa de transmissão de quem envia, limitados apenas pela janela de congestionamento. Caso a rede possua um roteador sobrecarregado, ou por alto uso de CPU ou por largura de banda indisponível, ele pode não ser capaz de processar esses pacotes em tempo hábil. Isso faz com que seu *buffer* encha e que pacotes sejam descartados. Consequentemente, quem está enviando entenderá que a rede está congestionada. Pelo algoritmo de controle de congestionamento do TCP, esse cenário implica, no mínimo, em retransmissão de pacote. Em casos mais extremos, implicará também em redução da janela de congestionamento. O resultado final é uma queda significativa de desempenho.

O protocolo QUIC propõe o uso de estimativas da largura de banda para definir intervalos de espera entre o envio de pacotes evitando, assim, exceder a largura de banda estimada. Esse mecanismo é chamado de *packet pacing*.

Experimentos com o Chrome mostraram que essa abordagem ajuda na redução da perda de pacotes. Em outras palavras, ela reduz a quantidade de pacotes descartados por roteadores devido à falta de espaço nos *buffers*. Os resultados mostraram que esse mecanismo reduz retransmissões em 25% e diminuem a quantidade de pacotes que demoram mais que o normal para serem entregues. Porém, ele não muda a mediana da latência para carregar uma página e não melhora a qualidade da experiência para vídeos no YouTube.

Controle de Congestionamento

Atualmente, o algoritmo de controle de congestionamento padrão no QUIC é uma reimplementação do TCP CUBIC. No código do QUIC disponibilizado pela Google também é possível trocar o controle de congestionamento do CUBIC para uma versão preliminar do BBR. A reimplementação do CUBIC faz os cálculos quantificando valores em bytes ao invés de pacotes. Para janelas de congestionamento pequenas, essa alteração implica que mais dados poderão ser enviados de um nó para o outro antes que ocorra um ajuste na janela. Por exemplo, se um servidor tem 5 pacotes para enviar e a janela de congestionamento atual é igual a 0,7, ele poderá enviar apenas $\lfloor 5 * 0,7 \rfloor = 3$ pacotes. Entretanto, se o cálculo fosse feito em bytes, o servidor poderia enviar 3,5 pacotes, ou seja, mais de 10% a mais de dados. É importante ressaltar que o ajuste da taxa de transmissão, tanto pela alteração da janela de congestionamento quanto pela espera maior no *packet pacing*, deve ser similar ao que é feito no TCP. Especificamente, o impacto geral deve ser equivalente ao que aconteceria no TCP se os fluxos multiplexados fossem substituídos por conexões TCP separadas e uma delas detectasse congestionamento. O objetivo é ser amigável ao TCP sem dominar os fluxos e impedir o TCP de funcionar, mas também sem perder controle e deixar que o TCP domine a utilização da rede.

O QUIC também oferece melhor sinalização que o TCP. Por exemplo, todo pacote, original ou retransmitido, usa números de sequência diferentes. Isso evita o problema de ambiguidade de retransmissão que o TCP possui, caracterizado pela impossibilidade de identificar se o ACK recebido é do pacote original ou do retransmitido. Já que não sabemos qual é o pacote correspondente do ACK recebido, não podemos atualizar os estimadores de RTT quando esse ACK finalmente chegar [KP87]. Além disso, como dados foram retransmitidos e o *exponential backoff*[1] foi aplicado, não podemos calcular um novo tempo para expirar o pacote e somos obrigados a usar o último tempo que calculamos para a próxima retransmissão. O efeito disso tudo é uma possível queda de desempenho do TCP.

2.3.5 SCTP *vs* QUIC

O SCTP (*Stream Control Transmission Protocol*), introduzido pela RFC 3286 [YO02] e especificado em detalhes pela RFC 4960 [Ste15], é um protocolo de transporte confiável que opera usando o protocolo IP. Apesar de ter sido projetado para transportar mensagens telefônicas em redes IP

[1] *Exponential backoff* significa que a mensagem é reenviada em intervalos exponencialmente crescentes.

(PSTN, *Public Switched Telephone Network*), ele também possui funcionalidades que o tornam apto para funcionar com uma gama maior de aplicações. Assim como o QUIC, o SCTP visa substituir o TCP, e muitos de seus mecanismos são semelhantes aos do TCP. Diferentemente do QUIC, ele não é construído para rodar sobre o UDP.

O SCTP fornece serviço confiável de transporte, ou seja, garante a entrega de pacotes em ordem e sem erros. Ele também é orientado a sessões, o que significa que uma "associação" tem que ser criada entre as duas pontas de uma conexão antes que dados possam ser transmitidos. Essa associação é mantida até que todos os dados tenham sido transmitidos corretamente. As transmissões no SCTP são *full duplex*, ou seja, dados podem ser enviados nas duas direções simultaneamente. O SCTP foi projetado para cooperar com o TCP no uso de recursos da rede. O mecanismo de reconhecimento de pacotes é igual ao TCP com SACK (*Selective Acknowledgement*) [FMMR13] com pequenas adaptações, ou seja, um pacote ACK possui um campo que representa intervalos de número de sequência. Esses intervalos indicam pacotes que ainda não foram recebidos. Os controles de fluxo e congestionamento também são equivalentes ao do TCP.

Ao contrário do TCP, o SCTP fornece um mecanismo para entrega de dados em fluxos independentes com o objetivo de evitar *head-of-line blocking*. Um mecanismo de *multi-homing* também é implementado, ou seja, uma mesma ponta de uma conexão pode ter múltiplos endereços IP. Essa funcionalidade melhora a tolerância a falhas da sessão porque diferentes IPs podem fazer com que pacotes sejam enviados por caminhos diferentes. Caso o IP principal não possa ser alcançado (por uma falha física na rede, por exemplo), o IP secundário do destino é usado. Além disso, o SCTP implementa um mecanismo de *cookies* para armazenar o estado da conexão.

A especificação não define protocolos e procedimentos de segurança. Para essa finalidade, o SCTP pode ser estendido para usar o TLS ou o DTLS. Usar o TLS com o SCTP como definido na RFC 3436 [JRT02], porém, traz algumas limitações para o SCTP. Dentre elas, as principais são:

- A parte que envia dados SCTP pode indicar que um pacote, ou até mesmo todos os pacotes de um fluxo, podem ser entregues fora de ordem. Porém, o TLS requer que a camada de transporte entregue seus registros em uma ordem específica. Portanto, não é possível utilizar a funcionalidade de entrega fora de ordem do SCTP;

- Apesar de o SCTP usar fluxos unidirecionais, o TLS exige que exista uma correspondência entre o que é pedido pelo cliente e o que é respondido pelo servidor. Por isso, são criados fluxos bidirecionais ao se usar o mesmo identificador para dois fluxos diferentes. Por conta disso, o número de conexões que podem ser abertas para cada associação fica limitado pelo número de fluxos bidirecionais e,

- Uma conexão TLS precisa ser aberta para cada fluxo bidirecional, causando perda de desempenho significativa se muitos fluxos são abertos.

Já o DTLS foi projetado para não alterar características do transporte de datagramas (como no UDP) [RM15]. Como não impede que segmentos sejam recebidos fora de ordem, não garante entrega e protege a conexão como um todo (e não cada fluxo individual), usar o DTLS supera essas limitações automaticamente.

Em alguns aspectos o SCTP é parecido com o QUIC, principalmente pelo fato de ambos implementarem multiplexação e pelo uso do DTLS que fornece criptografia e autenticação para o UDP. Por conta dessa semelhança e por já ser um padrão da IETF, pode-se perguntar porque se preferiu desenvolver um protocolo do zero do que usar o SCTP junto com o DTLS. A resposta está na latência para iniciar uma comunicação.

Um dos maiores ganhos do QUIC é no estabelecimento de conexão que tipicamente pode ser feito em 0-RTT, permitindo envio de dados já no primeiro pacote. Um dos fatores para que esse ganho seja alcançado é o fato de o QUIC unir o estabelecimento de conexão para transporte e para criptografia em apenas uma camada. Já o SCTP e DTLS funcionam em camadas diferentes, o que implica em estabelecimento de conexões separadas. Além disso, conexões DTLS precisam ser estabelecidas e terminadas na mesma associação SCTP e não podem abranger outras associações.

Isso impossibilita que o restabelecimento de conexão (com envio de dados) seja feito logo no primeiro RTT. Alterar os protocolos para reduzir o processo de estabelecimento de conexão é plausível, mas é improvável que ganhos de desempenho comparáveis aos do QUIC possam ser atingidos sem grandes mudanças nos protocolos, até mesmo sem uní-los. Finalmente, alterações no SCTP exigem atualizações nos sistemas operacionais, causando os mesmos problemas de implantação existentes no TCP. Escolhemos fazer a análise de desempenho no QUIC porque ele parece ter mais chances de ter sucesso e de ser mais usado que o SCTP.

Capítulo 3

Trabalhos Relacionados

Neste capítulo, apresentamos alguns trabalhos similares ao nosso. A nossa ideia é descrever pesquisas de análise de desempenho de protocolos de transporte em redes de computadores, com foco em trabalhos relacionados com a avaliação de diferentes algoritmos de controle de congestionamento. A literatura nesse tópico é vasta, principalmente em pesquisas sobre como melhorar o desempenho do TCP. Por isso, dividimos este capítulo em duas seções: uma que apresenta trabalhos de análise de desempenho do TCP e outra que descreve trabalhos de análise de desempenho do QUIC. É importante ressaltar que a ideia deste capítulo não é descrever a proposta de melhoria quando há uma. Ao invés disso, detalhamos como a análise de desempenho foi realizada: qual técnica foi utilizada, quais as métricas coletadas, quais as configurações e topologia de rede dos experimentos, os resultados obtidos e como eles foram validados, as limitações apresentadas e, por fim, quais as semelhanças e diferenças em relação a este trabalho. Porém, para facilidade de entendimento, resumimos as características marcantes de cada variante do TCP na Tabela 3.1. O que definimos apenas como TCP também pode ser chamado de TCP New Reno.

Variação	**Resumo**
BIC	Usa busca binária para encontrar a taxa de transmissão ideal sem congestionar a rede.
CUBIC	Usa uma função cúbica para controlar o tamanho da janela de congestionamento, com seu ponto de inflexão igual ao tamanho da janela quando ocorreu o último evento de perda de pacote.
Westwood	Baseado em estimativas de largura de banda para ajustar a janela de congestionamento.
Westwood+	É uma modificação do Westwood que evita superestimar a largura de banda na presença de congestionamento.
Tahoe	Na presença de 3 ACKs duplicados, reduz a janela de congestionamento para 1 MSS e retorna ao estado de partida lenta.
Reno	Na presença de 3 ACKs duplicados, reduz a janela de congestionamento para a metade do valor da janela quando houve congestionamento e entra no estado de recuperação rápida.
New Reno	É uma modificação do Reno que otimiza seu algoritmo de recuperação rápida.
BBR	Faz estimativas periódicas do RTT mínimo e da largura de banda no enlace gargalo com o objetivo de manter sua taxa de transmissão no ponto ótimo de operação da rede.

Tabela 3.1: *Resumo das variantes citadas do TCP.*

3.1 Trabalhos de Análise de Desempenho do TCP

O desempenho do TCP, por ser o protocolo de transporte mais utilizado por protocolos de aplicação da Internet e por ter papel fundamental para o funcionamento da rede, é tópico de muita pesquisa desde que foi criado. Nesta seção, selecionamos alguns trabalhos que ilustram as diferentes abordagens para analisar seu desempenho.

No trabalho em que o CUBIC é proposto [HRX08], os autores realizam uma modelagem analítica do algoritmo e por meio dela, mostram as vantagens de desempenho do CUBIC se comparado ao seu antecessor (o BIC) e ao TCP. Esses resultados foram validados através de medições em uma topologia Dumbbell (veja o Capítulo 5), com a implementação do CUBIC encontrada no Linux. Porém, apenas uma configuração de rede foi utilizada. Os autores estudam dois aspectos que não tiveram bom desempenho no BIC e que motivaram a construção do CUBIC:

- A **justiça** entre fluxos de mesmo protocolo tanto para fluxos com RTTs iguais quanto para fluxos com RTTs diferentes. Essa métrica é calculada como a razão das vazões de dois fluxos e representa o grau de compartilhamento de banda para fluxos de um mesmo protocolo e,
- O **impacto em um fluxo TCP**, calculado como a razão das vazões de um fluxo TCP e um fluxo de uma variante do TCP.

Os autores concluem que o CUBIC é mais "amigável" com fluxos TCP e que melhora a justiça de fluxos, não favorecendo aqueles de RTT menores.

[LCK14] apresenta uma implementação do CUBIC no simulador de redes NS-3. Essa implementação é baseada no código encontrado no Linux. A validação da implementação é feita comparando resultados de simulações do CUBIC no NS-3 com resultados de medições do CUBIC no Linux e com resultados de simulações de outra implementação do CUBIC no simulador NS-2. A topologia de rede que os autores usaram foi semelhante ao que chamamos de topologia simples nos nossos experimentos (veja o Capítulo 5). O módulo *Flow Monitor* do NS-3 foi utilizado para calcular a **quantidade de pacotes enviada e recebida** em um intervalo de tempo para determinar se a implementação do CUBIC no NS-3 se comportava da mesma forma que as outras implementações. O trabalho também compara o desempenho do CUBIC com o TCP, apresentando ganhos significativos de desempenho. As limitações do trabalho são que a implementação dos autores não usa *timestamps* como na implementação do Linux, fazendo com que o desempenho entre as duas seja um pouco diferente, e que as simulações foram feitas em apenas um cenário, sem variação de largura de banda e atraso, por exemplo.

Já [LSM08] apresenta um estudo de medições do desempenho do CUBIC. Nesse trabalho, a topologia de rede escolhida foi a de Dumbbell com um roteador adicional entre os dois tradicionais. Assim, os autores puderam emular diferentes cenários de rede ao variar a banda disponível e tempo de propagação no enlace. A implementação do CUBIC usada é a encontrada no Linux. Cada teste foi executado pelo menos 10 vezes ou rodaram por pelo menos 1 hora. Os autores analisaram as seguintes métricas:

- A **justiça** entre fluxos de mesmo protocolo tanto para fluxos com RTTs iguais quanto para fluxos com RTTs diferentes. Essa métrica foi analisada por meio de comparações da evolução do tamanho da janela de congestionamento e da largura de banda com o passar do tempo e razão das vazões entre dois fluxos;
- A **capacidade de utilização dos enlaces**, calculada como a razão entre a vazão média e o produto entre banda e atraso da rede em um fluxo. Valores próximos de 1 indicam que a utilização dos enlaces é alta e,
- **Retrocompatibilidade**, calculada pela razão das vazões de um fluxo CUBIC e um fluxo TCP.

Os resultados mostraram que esse algoritmo demora para convergir, usa melhor os recursos da rede na maior parte dos cenários e não é "amigável" com o TCP em algumas situações.

[GNUS13] apresenta uma implementação dos controles de congestionamento TCP Westwood e Westwood+ no NS-3. Ela é baseada no pseudocódigo disponibilizado nos trabalhos que introduziram essas duas variações do TCP. A validação da implementação é feita por meio de comparações de como a **vazão** se comporta (em cenários variando a taxa de erros, a largura de banda e o atraso) com os resultados reportados nesses trabalhos. Os autores usam duas topologias para os experimentos: uma em que uma fonte e um destino estão conectados por um roteador e outra, para corresponder exatamante à topologia usada no trabalho que introduziu o Westwood+, em que duas fontes estão conectadas a um servidor por meio de um roteador. O trabalho também compara o desempenho do TCP Westwood e Westwood+ com outras 3 variações do TCP: o Tahoe, Reno e New Reno. Tanto o Westwood quanto o Westwood+ tiveram desempenho melhor que os outros 3 TCP em todos os cenários. Os autores também apontam que mais cenários para experimentos e validação com estudos de medição são necessários.

No trabalho em que o TCP Westwood é proposto [CGM+02], os autores apresentam uma maneira de estimar a largura de banda disponível e de como usá-la para ajustar a janela de congestionamento de forma mais apropriada se uma perda de pacote for detectada. Eles estudam as alterações por meio de modelagem analítica, mostrando que o TCP Westwood terá desempenho melhor que o TCP Reno e que haverá justiça entre fluxos desse novo algoritmo. Os resultados da modelagem são validados com simulações no NS-2 e por medições usando uma implementação deles do protocolo no Linux. Nas simulações, os autores utilizaram duas topologias: uma em que duas fontes conectadas a um destino comum compartilham o mesmo enlace gargalo e outra em que uma fonte está conectada a um destino em uma rede mista (parte cabeada e parte sem fio). As métricas calculadas foram:

- A **eficácia** das estimativas de banda, analisada pelo comportamento do tamanho da janela de congestionamento;
- A **justiça** entre fluxos TCP Westwood, analisada pela evolução dos números de sequência durante uma conexão;
- O **impacto** em fluxos TCP Reno, estudado a partir da vazão média dos fluxos e,
- A **vazão**, definida pela razão entre a quantidade de dados enviada por segundo.

Os resultados das simulações mostraram que a estimativa de banda de fato é eficaz, que o Westwood é justo, que não tem impacto tão grande em (não impede o funcionamento de) fluxos TCP Reno e que o desempenho do Westwood é melhor, principalmente em redes sem fio. Nas medições, clientes estavam conectados a servidores acessíveis pela Internet. Nelas, foram calculadas a **vazão**, **justiça** entre fluxos TCP Westwood e **impacto** em fluxos TCP Reno. Os resultados das medições corroboraram o que foi observado nas simulações.

[MGF+04] segue uma abordagem similar para estudar o desempenho do TCP Westwood+. Os autores, por meio de modelagem analítica, mostram o ganho de desempenho de uma nova forma proposta de estimar a largura de banda, que evita que o TCP Westwood+ tenha tanto impacto em fluxos TCP Reno quanto o TCP Westwood. Nesse trabalho, esse resultado foi validado por meio de medições e simulações no NS-2. Nas medições, usaram uma implementação do protocolo que escreveram para o Linux e duas topologias de rede: uma em que 2 clientes estão conectados a 1 servidor por meio de um roteador (que é usado para emular atrasos) e outra em que 2 clientes estão conectados a 1 servidor acessível pela Internet. Em ambas as medições, foram calculados o **goodput**, definido como a razão entre o tamanho do arquivo transferido e o tempo de transferência, e a **eficácia** das estimativas de largura de banda, avaliada a partir da evolução do tamanho da janela de congestionamento ao longo de uma conexão. Nas duas medições, o TCP Westwood+ se mostrou eficaz para estimar a banda disponível além de ter melhor *goodput* se comparado ao TCP Reno. Nas simulações, diversas topologias de rede foram usadas como Dumbbell com números diferentes de

clientes e servidores e redes mistas com partes cabeadas e partes sem fio. Nas simulações, **goodput** e **justiça** de fluxos são calculados. A justiça foi estudada pela comparação de estimativas de largura de banda e evolução dos números de sequência durante uma conexão. Os resultados corroboraram as conclusões da modelagem analítica e das medições.

[LLS07] apresentam resultados de experimentos de medição de desempenho de diversas variações do TCP. Utilizaram as implementações existentes no Linux para a análise. No trabalho, a topologia Dumbbell foi utilizada com uma pequena modificação: um roteador adicional foi posicionado entre os dois roteadores das pontas. Com isso, os autores podiam emular diferentes condições de rede, nesse roteador central, ao variarem o tamanho dos *buffers*, banda disponível e tempo de propagação. As métricas calculadas foram:

- A **justiça** tanto para fluxos de mesmo RTT quanto para fluxos com RTTs diferentes. Para avaliarem a justiça, calculam as médias de vazão dos fluxos;
- A **retrocompatibilidade** das variantes do TCP com o TCP, usando os mesmos cálculos para analisar a justiça;
- A **utilização da capacidade dos enlaces**, comparando a vazão média com o produto entre a largura de banda do enlace gargalo e o atraso do caminho de um pacote e,
- A **sensibilidade**, ou capacidade de adquirir e liberar banda conforme as condições de rede mudam, avaliada de duas formas: pelos tempos de resposta e convergência. Para avaliar o tempo de resposta, é calculada a vazão média de um fluxo conforme perdas de pacotes aleatórias acontecem. Para avaliar o tempo de convergência, medem o tempo para que os fluxos da rede alcancem a mesma vazão média quando um novo fluxo entra na rede.

Os resultados mostraram que a maioria das variações testadas não apresentam justiça na utilização da rede, que todas melhoram a utilização da capacidade dos enlaces em grande parte dos cenários testados, que o TCP é o menos sensível às perdas de pacotes e que o TCP converge mais rápido que a maioria das variações.

[HBZ17] analisa o comportamento do BBR em situações específicas e fornece experimentos de medição do algoritmo. As métricas calculadas foram:

- O **atraso em filas**, definido como o tempo que um pacote espera até ser processado e redirecionado por um roteador;
- A **quantidade de perdas de pacote** que ocorrem durante uma transmissão;
- A **vazão**, definida como a quantidade de bits enviada por segundo;
- O **goodput**, definido como a quantidade de megabits do arquivo transferido por segundo, excluindo retransmissões e,
- A **justiça**, tanto para fluxos de mesmo RTT quanto para fluxos de RTTs diferentes, avaliada por comparações de *goodput* e vazão dos fluxos.

Usaram a topologia Dumbbell para os experimentos. Os autores eram capazes de emular atraso na propagação em um dos roteadores e tinham controle do tamanho de seu *buffer*. Eles também utilizaram a implementação do BBR disponível no Linux. Cada experimento foi repetido pelo menos 5 vezes. Os resultados mostraram que, para um fluxo BBR sozinho, o comportamento do algoritmo é o esperado. Porém, a taxa de entrega do algoritmo diminui quando múltiplos fluxos tem o mesmo enlace gargalo, devido ao aumento do atraso nas filas. Esse aumento nas filas causa grandes quantidades de perdas de pacote. Por fim, ao comparar o desempenho do BBR com o CUBIC, os autores reportam que o BBR apresenta melhor vazão, mas que não é justo com fluxos CUBIC. Além disso, ao contrário do CUBIC, o BBR não reage a perdas de pacote. No caso em que múltiplos fluxos BBR fazem com que muitos pacotes sejam descartados, os fluxos CUBIC detectam congestionamento e

reduzem sua taxa de transmissão, enquanto que os fluxos BBR ignoram essas perdas e continuam enviando pacotes a uma alta taxa de envio.

A Tabela 3.2 mostra uma comparação entre os trabalhos apresentados, as técnicas de análise de desempenho utilizadas (indicadas pela letra T) e quais delas foram usadas para validação, se houve (indicadas pela letra V). Os trabalhos que sugerem algoritmos novos de controle de congestionamento normalmente os apresentam com uma modelagem analítica e usam simulações e/ou medições para validar os resultados. Os que analisam o desempenho de variações do TCP por meio de simulações geralmente validam os resultados com medições e, às vezes, com simulações realizadas em um simulador diferente. Utilizar uma técnica para validar os resultados obtidos por outra é forma recomendada por [Jai90] (veja a seção 2.1) para confirmar que a análise está correta. Os trabalhos de medição que estudamos, por sua vez, não validam os resultados. No nosso trabalho, a abordagem de implementação no simulador que usamos foi diferente da abordagem dos trabalhos apresentados nesta seção: nós portamos o código do QUIC para o NS-3 ao invés de implementar o protocolo inteiro (mais detalhes no Capítulo 4). Nossa validação foi feita por comparações do tempo total de transmissão de bytes com resultados de medição publicados por [KJC+17].

	Modelagem Analítica	**Simulação**	**Medição**
[CGM+02]	T	V	V
[MGF+04]	T	V	V
[LLS07]	-	-	T
[HRX08]	T	-	V
[LSM08]	-	-	T
[GNUS13]	V	T/V	V
[LCK14]	-	T/V	V
[HBZ17]	T	-	V
Este trabalho	**-**	**T**	**V**

Tabela 3.2: *Comparação entre os trabalhos relacionados à análise de desempenho do TCP apresentados de acordo com as técnicas e formas de validação utilizadas.*

A Tabela 3.3, por sua vez, compara as métricas de desempenho utilizadas em cada trabalho apresentado. Podemos notar que diversas métricas são usadas para analisar o desempenho do TCP. No nosso trabalho, além do tempo de transmissão de dados, analisamos a justiça entre fluxos QUIC e o impacto do QUIC na taxa de transmissão de um fluxo TCP, as métricas mais usadas nos trabalhos relacionados ao TCP. Normalmente esse impacto (ou retrocompatibilidade) também é chamado de justiça porém, como alguns trabalhos estudados não fazem essa análise, a separação em duas métricas diferentes foi necessária. É importante destacar que o nosso trabalho é o único que compara o tempo de transmissão.

	[CGM+02]	[MGF+04]	[LLS07]	[HRX08]	[LSM08]	[GNUS13]	[LCK14]	[HBZ17]	**Este trabalho**
Justiça	X	X	X	X	X	-	-	X	**X**
Impacto	X	-	X	X	X	-	-	-	**X**
Pacotes enviados	-	-	-	-	-	-	X	-	**-**
Pacotes recebidos	-	-	-	-	-	-	X	-	**-**
Utilização de banda	-	-	X	-	X	-	-	-	**-**
Vazão	X	-	-	-	-	X	-	X	**-**
Eficácia de estimativas	X	X	-	-	-	-	-	-	**-**
Goodput	-	X	-	-	-	-	-	X	**-**
Sensibilidade	-	-	X	-	-	-	-	-	**-**
Atraso em filas	-	-	-	-	-	-	-	X	**-**
Perdas de pacotes	-	-	-	-	-	-	-	X	**-**
Tempo de transmissão	-	-	-	-	-	-	-	-	**X**

Tabela 3.3: *Comparação entre os trabalhos relacionados à análise de desempenho do TCP apresentados e as métricas de desempenho utilizadas.*

3.2 Trabalhos de Análise de Desempenho do QUIC

O protocolo QUIC ainda não possui uma especificação definida e utiliza algoritmos de controle de congestionamento que também já foram integrados ao TCP. Além disso, a única implementação amplamente utilizada é a da Google. Outras implementações existem, mas elas são apenas o mesmo código reescrito em outra linguagem de programação e são usadas em servidores de produção. Nenhuma delas foi feita para simuladores. Por essas razões, todos os trabalhos que encontramos de análise de desempenho do QUIC são de medição.

[BG16] realiza medições para comparar o desempenho entre o HTTP/2 e o QUIC, usando como métrica a **razão entre o tempo de carregamento de páginas web estáticas do QUIC e do HTTP/2**. Dessa forma, se carregar uma página usando o QUIC for mais rápido, a razão será maior que 1. O tempo de carregamento foi calculado como o tempo entre o início da transmissão até o recebimento do último byte. Para levar em consideração variações na rede, os autores fizeram 5 medições para cada página e calcularam a mediana do tempo de *download*. Dois cenários foram considerados: um controlado, no qual um cliente estava conectado a um servidor por meio de um roteador e um não controlado, no qual o servidor estava acessível pela Internet. Além disso, eles usaram o `Netem` [Net] para emular diferentes condições de rede como largura de banda, atraso e taxa de perda de pacotes. Os resultados mostraram que o QUIC teve desempenho melhor em redes com pouca largura de banda, alto atraso e alta taxa de perda de pacotes, mas que teve desempenho pior em páginas compostas de muitos objetos de tamanho pequeno. As limitações dessa medição são que o servidor do QUIC disponível para uso não é preparado para rodar em produção e que não fizeram experimentos em páginas web dinâmicas.

[CDCM15] também realiza medições para avaliar o desempenho do QUIC em duas situações: uma em comparação com o CUBIC e outra em comparação com o SPDY (protocolo que serviu de base ao HTTP/2) e o HTTP/1.1. Em ambas, os servidores estavam acessíveis pela Internet. A implementação de cliente e servidor do QUIC disponibilizada pela Google como biblioteca *standalone* foi utilizada. Os autores desenvolveram uma ferramenta para regular a largura de banda e o atraso no cliente. Eles consideraram 4 métricas: ***goodput***, calculada como a taxa média de recebimento (sem considerar retransmissões); **utilização do canal**, definida como a razão entre a taxa média de recebimento e a capacidade do canal; **razão de perda**, definida como a razão entre a quantidade de bytes perdida e a quantidade de bytes enviada e o **tempo de carregamento de uma página**, calculada como o tempo gasto para o navegador fazer *download* de uma página e processar todos os seus objetos. Os resultados mostraram que o QUIC tem maior *goodput* que o CUBIC, que

reduz o tempo de carregamento de uma página web se comparado ao SPDY e o HTTP/1.1 e que o desempenho do QUIC piora quando o FEC (veja a subseção 2.3.4) está habilitado. As limitações do trabalho são que a implementação utilizada do QUIC não é feita para rodar em produção e que fizeram experimentos apenas com *downloads* de páginas estáticas.

Em [LRW+17], os desenvolvedores do QUIC reportam resultados de medições de desempenho feitas usando os servidores de produção da Google e grupos de teste no *Chrome*. Os resultados foram comparados com o desempenho do TCP com TLS. Tanto o QUIC quanto o TCP usavam o CUBIC para controle de congestionamento. Nesse trabalho, 3 métricas foram calculadas: latência de busca, latência de reprodução de vídeo e taxa de *rebuffer*. A **latência de busca** é definida como o tempo entre o envio de uma requisição de busca até que todo o conteúdo seja apresentado para o cliente. A **latência de reprodução de vídeo** é calculada como o tempo entre um usuário clicar para o vídeo começar e o vídeo, de fato, iniciar. A **taxa de *rebuffer*** é a porcentagem de tempo que um vídeo pausa durante sua reprodução para armazenar dados em *buffers*. O QUIC teve desempenho melhor que o TCP com TLS nas 3 métricas, na maioria dos casos. O QUIC pode ter desempenho pior em redes com alta largura de banda, pouco atraso ou pouca taxa de perda de pacotes. O desempenho do QUIC também piora para usuários com dispositivos móveis. Nesse trabalho, alguns detalhes sobre os cenários de rede são apresentados. Por exemplo, todos os gráficos apresentados mostram resultados, como a latência, em função do RTT mínimo observado na conexão. Porém, detalhes importantes sobre os cenários, como quantidade de dados enviada e a largura de banda no enlace gargalo, não são detalhados. A implementação de servidor do QUIC utilizada também possui otimizações que não estão presentes na versão de código aberto. Por essas razões, reproduzir esses experimentos não é possível. No nosso trabalho, as simulações descrevem todas as características da rede tornando, assim, os experimentos reprodutíveis.

Já [KJC+17] descreve os experimentos de medição mais recentes (do final de 2017) do QUIC. Neles, um cliente é conectado à Internet por meio de um roteador que os autores controlam. Para emular diferentes cenários de rede, esse trabalho usa as ferramentas `Linux Traffic Control` [TC] e `Netem` no próprio roteador. Dessa forma, eles podem controlar a banda disponível, perda de pacotes, atraso, *jitter* e reordenação de pacotes. Fazem medição em duas situações: uma de *download* de uma página web estática (que pode ser composta por um número variável de objetos) e outra de *streaming* de vídeo no YouTube. Os autores calculam a **vazão**, o **tempo de carregamento de uma página** (definido como o tempo para carregar todos os objetos de uma página), **justiça** entre fluxos QUIC e entre o QUIC e o TCP e métricas relacionadas à qualidade de vídeo como o **tempo para iniciar** e **eventos de *rebuffering***. Nesse trabalho, a análise de desempenho é feita entre o HTTP/2 rodando sobre o QUIC e o HTTP/2 utilizando TCP CUBIC com TLS. Cada medição foi feita 10 vezes pelo menos. A implementação de servidor do QUIC usada não é feita para rodar em produção. Para superar esse problema, os autores calibraram os parâmetros da versão disponível do QUIC até eles conseguirem desempenho parecido com os servidores da Google. Os resultados mostraram que o QUIC carrega mais rápido páginas web na maior parte dos cenários de rede e que tem desempenho melhor nas métricas de qualidade de vídeo, principalmente para vídeos de alta resolução. Porém, o QUIC não é justo com fluxos TCP. Em cenários com *downloads* de páginas web compostas por muitos objetos pequenos, o TCP obteve resultados melhores e o desempenho do QUIC piora significativamente quando muitos pacotes chegam fora de ordem. A maior parte dos resultados foram validados pelas medições reportadas pela Google. Porém, para alguns cenários, como o de alta reordenação de pacotes, não há outros trabalhos que reportaram resultados em condições de rede similares e, portanto, a validação dos resultados desse trabalho é parcial. Como esse trabalho é o mais recente de medição com o QUIC e usa uma versão do código similar à que usamos, além de detalhar parâmetros dos experimentos, utilizaremos seus resultados para validar os resultados dos nossos experimentos.

A Tabela 3.4 mostra uma comparação entre os trabalhos apresentados, as técnicas de análise de desempenho utilizadas (indicadas pela letra T) e quais delas foram usadas para validação, se houve (indicadas pela letra V). Podemos notar que a maior parte dos trabalhos não validam seus resultados e que nenhum deles usa resultados obtidos por outro método de análise de desempenho,

como recomendado por [Jai90] (veja a seção 2.1). Neste trabalho, realizamos a primeira análise de desempenho do QUIC por meio de simulações. Nossos resultados, além de terem sido validados por medições já existentes, também podem ser usados por outros trabalhos que visam analisar o desempenho do QUIC através de modelagem analítica ou medições.

	Modelagem Analítica	**Simulação**	**Medição**
[CDCM15]	-	-	T
[BG16]	-	-	T
[LRW+17]	-	-	T
[KJC+17]	-	-	T/V
Este trabalho	**-**	**T**	**V**

Tabela 3.4: *Comparação entre os trabalhos relacionados à análise de desempenho do QUIC apresentados de acordo com as técnicas e a forma de validação utilizadas.*

A Tabela 3.5, por sua vez, compara as métricas de desempenho utilizadas em cada trabalho apresentado. É importante notar que, apesar de os trabalhos usarem nomes diferentes, algumas das métricas são calculadas da mesma forma ou têm a mesma semântica. Por exemplo, a latência de busca usada em [LRW+17] tem a mesma conotação do tempo de carregamento dos outros trabalhos: representa o tempo passado entre o cliente fazer uma requisição e receber toda a resposta do servidor. O mesmo vale para a latência de reprodução e taxa de *rebuffer* de [LRW+17], equivalentes ao tempo para iniciar vídeos e eventos de *rebuffering* de [KJC+17], respectivamente. Neste trabalho, analisamos o desempenho do QUIC por 2 métricas: justiça (tanto entre fluxos QUIC quanto entre um fluxo QUIC e outro TCP) e o tempo de carregamento. No nosso contexto de simulações no NS-3, porém, não há de fato o carregamento de uma página web. Nas simulações há apenas a transmissão de dados de um nó para o outro. Por isso, neste texto denotamos essa métrica de **tempo total de transmissão de dados**. Essas métricas, nossas topologias e configurações de rede serão discutidas no Capítulo 5.

	[CDCM15]	[BG16]	[LRW+17]	[KJC+17]	**Este trabalho**
Tempo de Carregamento	X	X	X	X	**X**
Goodput	X	-	-	-	**-**
Utilização do Canal	X	-	-	-	**-**
Razão de Perda	X	-	-	-	**-**
Latência de Reprodução	-	-	X	X	**-**
Rebuffers	-	-	X	X	**-**
Vazão	-	-	-	X	**-**
Justiça	-	-	-	X	**X**

Tabela 3.5: *Comparação entre os trabalhos relacionados à análise de desempenho do QUIC apresentados e as métricas de desempenho utilizadas.*

Capítulo 4

Módulo do QUIC no NS-3

Neste capítulo, apresentamos os detalhes técnicos de implementação do módulo do QUIC no NS-3 [ns3], descrevendo o processo que utilizamos, as dificuldades encontradas e soluções para superá-las e como verificamos que o módulo realmente funciona.

Nós tínhamos algumas opções para implementar o protocolo no NS-3, relacionadas tanto ao local dentro da infraestrutura do simulador onde encaixaríamos o código quanto à forma com que escreveríamos todas as funcionalidades necessárias do QUIC.

A arquitetura do NS-3 segue o modelo TCP/IP e sua base de código contém as funcionalidades básicas para simular uma rede de computadores. Funcionalidades adicionais são integradas à base principal como módulos, de forma a ficarem disponíveis para uso nas simulações. Seguindo a arquitetura do NS-3, criamos um novo módulo que adiciona o protocolo à sua estrutura principal. Por se tratar de um protocolo de transporte que é implementado usando outro protocolo de transporte como base, poderíamos posicionar o módulo tanto no mesmo "nível" dos outros protocolos de transporte quanto como uma nova aplicação do NS-3. Uma aplicação no NS-3 não é similar a protocolos de aplicação como o HTTP, por exemplo. Uma aplicação no NS-3 é simplesmente uma classe que controla quando enviar bytes e o que fazer quando receber bytes da camada de transporte. Por ser mais parecido com como é implementado em produção, optamos por posicionar o módulo como uma **aplicação** do NS-3.

O QUIC usa o UDP e implementa todas as funcionalidades que precisa, como a garantia de entrega e o estabelecimento de conexão. Assim como no Linux, o NS-3 usa *sockets* como estruturas que abstraem o TCP e o UDP. Se quiséssemos usar os *sockets* do UDP, teríamos que implementar todas essas funcionalidades que os *sockets* TCP já fornecem. Como o NS-3 possui uma classe que move os dados entre a camada de transporte e a camada de rede, decidimos ter uma implementação própria de *sockets* TCP que os altera para as nossas necessidades e fazer a conexão com a camada de rede usando a classe do UDP para manter seu comportamento geral. Porém, como o QUIC ainda não está bem definido, essa opção se torna ruim. Basear nossa implementação nos *drafts* da IETF disponíveis não é suficiente pois diversos aspectos de funcionamento ainda não estão especificados. Decidimos, então, basear nossa implementação em uma versão da implementação da Google. Entretanto, nenhuma versão possui uma máquina de estados ou qualquer documentação que permita ser mais simples replicar suas funcionalidades. Como já estávamos estudando o próprio código para poder replicá-lo, optamos por integrar o código da Google diretamente no NS-3. A ideia foi identificar quais funções e classes eram necessárias para que um cliente e um servidor QUIC funcionassem no NS-3 e, a partir daí, integrar e adaptar as dependências. O QUIC é implementado dentro do código fonte do *Chromium*, mas a Google disponibilizava uma biblioteca *standalone* para o QUIC que não é mais atualizada e será removida. Por isso, fizemos um *fork* da última atualização dessa biblioteca [pro], que equivale à versão 41 do protocolo, e baseamos a implementação do módulo para o NS-3 (versão 3.27) nela. É importante notar que essa versão do código que usamos para o módulo não possui diferenças significativas com a versão mais atualizada que está no código do *Chromium*, pois o desenvolvimento está quase parado na espera de definições do grupo da IETF. Por fim, usar o próprio código que está no *Chromium* traria dificuldades na implementação do

cliente e servidor, que já estão prontos na biblioteca *standalone*.

A Figura 4.1 resume as atividades realizadas com relação à criação do módulo, detalhadas nas subseções 4.1 e 4.2. Nesta figura, é representada uma linha do tempo aproximada que coincide com o início e a finalização das atividades. Cada círculo tem diâmetro diretamente proporcional ao tempo gasto nas tarefas apontadas neles.

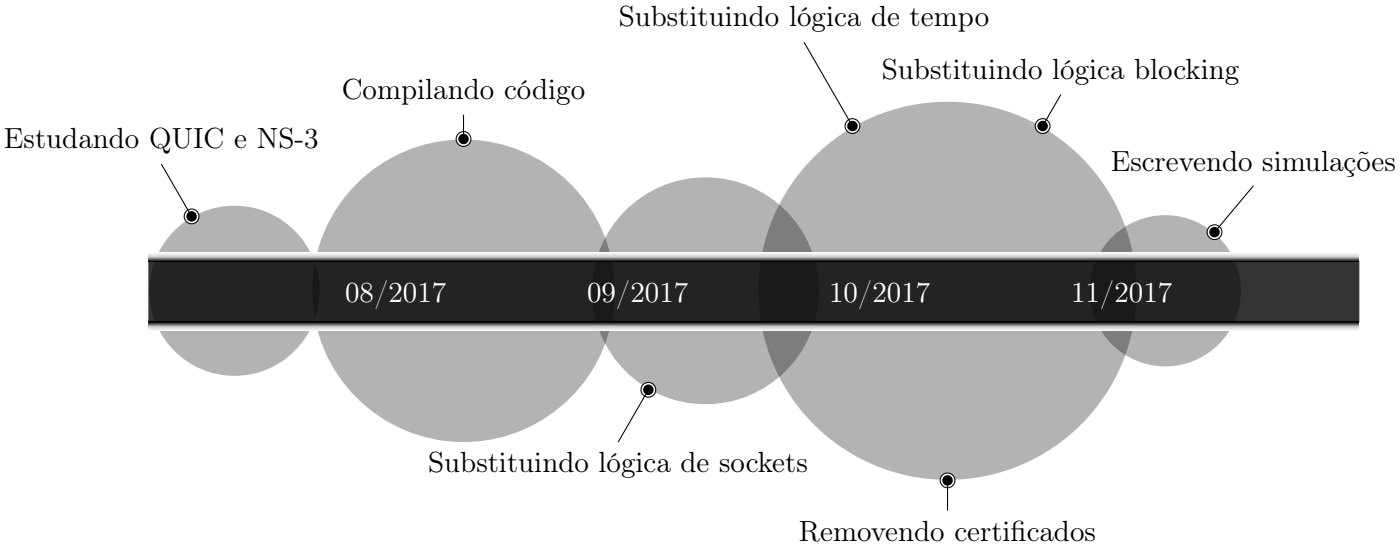

Figura 4.1: *Linha do tempo aproximada do tempo gasto nas tarefas de integração e adaptação do código do QUIC no NS-3.*

4.1 Detalhes Técnicos

Ao escrever as aplicações de cliente e servidor, usando funções do código do QUIC, tivemos que passar o código do QUIC para o nosso módulo do NS-3. A base de código é grande, tendo mais de quatro mil arquivos de implementação e vários outros projetos terceiros somados a ela. Além disso, enquanto o NS-3 usa o `Waf` [waf] como ferramenta de compilação de código, o QUIC usa `Gn` [GN] e `Ninja` [nin], ferramentas criadas para manutenção do Google Chrome. Assim, apesar de ambos os projetos estarem escritos em C++, não é direta a forma de integrar um no outro.

Uma solução considerada foi copiar todo o código fonte do QUIC para o módulo e tentar compilá-lo usando o `Waf`. Essa opção foi descartada pois algumas funções ou variáveis são implementadas em mais de um arquivo, causando erros de ligação. Escolhemos, então, os arquivos (com extensão `.h`) que declaram as funções que nós utilizamos e as implementações (arquivos com extensão `.cc`) relacionados a eles. Logo percebemos que estes arquivos escolhidos dependiam de outros arquivos, e estes de outros, e assim por diante.

Usando opções de compilação especiais do compilador `g++` (`-M` e `-MM`), que permitem listar as dependências (inclusões de cabeçalhos `.h`) de certo arquivo, escrevemos um programa que automaticamente importa todas as dependências e arquivos `.cc` de mesmo nome. Algumas implementações das funções que precisávamos, entretanto, não estavam em arquivos `.cc` de mesmo nome, principalmente relacionadas às funções que servem como uma interface e podem ter múltiplas implementações. Por exemplo, estas múltiplas implementações poderiam ser para sistemas operacionais diferentes. Nesses casos, bastou escolher a implementação para Linux, o sistema operacional utilizado em todos os experimentos. Em outros casos, as implementações estavam em arquivos que simplesmente tinham nomes diferentes e, por isso, tiveram que ser buscados manualmente.

Após esse processo, a implementação de alguns arquivos ainda não tinha sido encontrada. Descobrimos que o `Gn` gerava alguns arquivos automaticamente e então geramos estes arquivos com as implementações corretas. Alguns arquivos ainda estavam sendo compilados de maneira incorreta

devido à falta de opções de compilação, que seriam passadas pelo `Ninja`. Depois de algum estudo desses arquivos, conseguimos deduzir as opções necessárias e compilá-los.

Nesse momento, conseguimos compilar o cliente e servidor que tínhamos escrito. Porém, erros de ligação apareceram principalmente por causa de variáveis e funções sem implementação. Estes erros ocorriam porque alguns arquivos de bibliotecas dinâmicas (com extensão `.so`) faltavam ser incluídos em tempo de ligação. Os arquivos `.so` podiam ser gerados usando o `Ninja` e, após um estudo cuidadoso de quais arquivos incluir (já que, incluindo todos passávamos a ter erros de múltiplas implementações de variáveis e funções), conseguimos fazer a ligação dos objetos e gerar um executável. O código do QUIC, porém, continuava a fazer as chamadas de função usuais para lidar com *sockets* do sistema, ou seja, usava as funções da biblioteca `sys/socket.h`.

4.2 Adaptação do Código do QUIC

Como próximo passo, era necessário modificar o código do QUIC para usar os *sockets* UDP do NS-3 ao invés dos *sockets* UDP do sistema, cujas funções estão definidas em `sys/socket.h`. Para isso, fizemos uma cópia dessa biblioteca e reimplementamos todas as suas funções para usar os *sockets* UDP definidos no NS-3. Foi necessário estudar o código de *sockets* do NS-3 e a implementação de *sockets* no Linux para termos certeza de que o código do QUIC manteria o comportamento esperado. Com isso, mudamos todas as chamadas de função do QUIC a `sys/socket.h` para, na verdade, chamar a nossa biblioteca de *sockets*. Assim, os *sockets* criados e utilizados pelo QUIC passaram a ser os do NS-3.

Outra aspecto que tivemos que adaptar é que a lógica de cliente e servidor do QUIC assume que o tempo é contínuo. O servidor, por exemplo, fica em um laço infinito esperando por pacotes vindos do cliente. Além disso, as funções que compõem cliente e servidor no QUIC nunca devolvem o controle para o código que as invocou, o que conflita com a lógica de tempo discreto de uma simulação no NS-3. Nele, uma simulação é baseada em eventos e, ao invés de o servidor esperar em um laço infinito por um pacote vindo do cliente, ele deve registrar uma função que será chamada quando um pacote chega. Esse tipo de função é chamado de função de *callback*. Quando o servidor receber um pacote, essa função for invocada e ela fizer o que for preciso, ela deve retornar controle ao NS-3 para que ele possa então chamar o próximo evento. Para isso funcionar da maneira correta, modificamos o código do cliente e servidor QUIC, trocando chamadas de funções que bloqueiam até receber algum pacote e laços infinitos por funções de *callback*. Também alteramos o sistema de alarme do QUIC para usar a lógica de tempo discreto do NS-3. Ao invés do QUIC esperar o tempo passar para disparar um alarme (por exemplo, de que algum pacote excedeu seu tempo de *timeout* e sem que um ACK tenha sido recebido), agora esse alarme é implementado como um evento que será escalonado para ser executado em algum momento no futuro.

Por fim, como não influenciam no comportamento da rede, removemos as partes do QUIC que tratavam de autenticação por certificados. Essa autenticação é usada no estabelecimento de conexão entre cliente e servidor, mas mantê-las deixaria a integração mais complexa por conta de outras dependências de bibliotecas externas que teriam que ser tratadas. Apesar da remoção dessa parte do código, o tempo de simulação não é afetado porque mantivemos toda a troca de pacotes para o estabelecimento de conexão.

Capítulo 5

Experimentos

Usamos duas topologias de rede para testarmos a integração do módulo do QUIC no NS-3 e avaliarmos o desempenho do protocolo: uma que chamamos de "simples" e outra de Dumbbell. As duas topologias estão ilustradas na Figura 5.1 e na Figura 5.2, respectivamente. A topologia simples, serve para avaliar o desempenho do QUIC em situações "ótimas", ou seja, sem levar em conta perdas e nós intermediários, simulando uma conexão física direta entre dois computadores ou uma rede descongestionada. Nas nossas simulações a transferência ocorre de B (servidor) para A (cliente).

Figura 5.1: *Topologia simples com 2 nós.*

Já a topologia Dumbbell serve para avaliar o desempenho dos protocolos em um cenário onde há tráfego de interferência. Cada nó Ai (cliente) recebe dados do nó Bi (servidor). Os nós R1 e R2 são roteadores em linha que conectam os clientes e servidores. A presença deles pode causar perda de pacotes se os *buffers* não tiverem espaço suficiente e se a quantidade de pacotes que chegam no roteador por unidade de tempo é maior do que a quantidade de pacotes que o roteador consegue processar e encaminhar por unidade de tempo. A perda de pacotes exercita uma parte fundamental dos protocolos de transporte: a identificação e reação ao congestionamento da rede. Essa é a principal diferença entre a topologia Dumbbell e a simples. A não ser quando explicitamente mencionado, a configuração padrão usada nas simulações é de 2 clientes e 2 servidores.

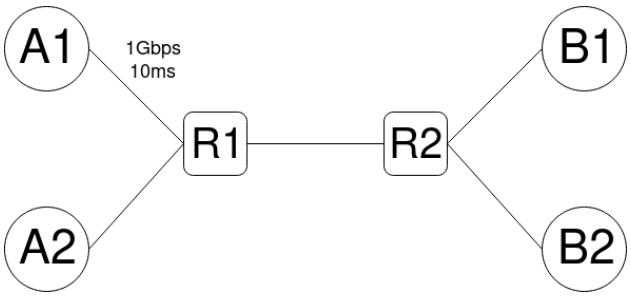

Figura 5.2: *Topologia Dumbbell com 4 nós (A1, A2, B1 e B2) e 2 roteadores (R1 e R2).*

A partir dessas topologias, criamos vários cenários de simulação ao variarmos, um por vez, os fatores largura de banda, atraso, taxa de perdas de pacotes e quantidade de dados enviada. Os valores utilizados para esses fatores são apresentados na Tabela 5.1. Eles foram escolhidos de forma a se aproximar ao máximo dos valores utilizados nas medições de [KJC+17]. Vale lembrar que os resultados de medição desse trabalho serão usados para validação tanto do módulo quanto das simulações. Por isso, é interessante que as topologias e as configurações de rede simuladas estejam

próximas do que foi utilizado nesse estudo. Os valores padrão de cada fator estão destacados em negrito. No caso da topologia Dumbbell, os fatores são variados no enlace entre os dois roteadores e a largura de banda e o atraso dos enlaces entre nós e roteadores são fixos em 1Gbps e 10ms, respectivamente. Para simularmos perdas de pacotes por conta de falhas no meio físico, usamos um modelo de erros fornecido pelo NS-3, o *RateErrorModel*. Esse modelo introduz erros em pacotes de acordo com uma distribuição de probabilidade que, no nosso caso, é uma **distribuição uniforme** entre 0 e 1. Dessa forma, quando atribuirmos, em um enlace, uma taxa de erros igual a 0,01, por exemplo, em torno de 1% dos pacotes serão perdidos nele.

Com relação às configurações de *hardware* e *software*, utilizamos um computador Intel® Core™ i7-2700K de 3,50GHz com 4 núcleos e com 16GB de memória RAM. O sistema operacional foi o Linux Ubuntu 16.04 com o kernel 4.13.0. Além disso, usamos a versão 3.27 do NS-3 e a versão 41 do QUIC. É importante notar que o QUIC está, até o momento desta escrita, na versão 44. Porém, não há diferenças significativas de funcionamento. As versões 42 e 43 permitem o recebimento de dados de fluxos que podem se sobrepor e adicionam um novo tipo de pacote, respectivamente. A versão 44 é uma implementação inicial do protocolo definido pelas *drafts* da IETF.

Fator	**Valores Testados**
Largura de banda	5Mbps, **10Mbps**, 50Mbps, 100Mbps, 500Mbps, 1Gbps
Atraso (ms)	10, **50**, 100, 150, 200
Taxa de erros	**0**, 0,0001, 0,001, 0,01, 0,1
Bytes transmitidos (KB)	10, 100, 200, 500, **1.000**, 10.000, 210.000

Tabela 5.1: *Valores dos fatores que usamos nas simulações. Os valores padrão estão em negrito.*

Para simular fluxos TCP, usamos as aplicações *BulkSend* e *PacketSink* do próprio NS-3. Essas aplicações funcionam apenas enviando dados na maior taxa de transmissão possível e recebendo os dados, respectivamente. Usamos todas as variações do TCP disponíveis no NS-3 para comparar o desempenho dos protocolos. No total, encontramos 13 variações diferentes: TCP NewReno, TCP BIC, TCP CUBIC, TCP Vegas, TCP HighSpeed, TCP Htcp, TCP Hybla, TCP Illinois, TCP Ledbat, TCP Scalable, TCP Veno, TCP Westwood e TCP Yeah. Calculamos duas métricas: **tempo total de transmissão dos dados** e **justiça**.

O tempo total de transmissão de dados foi calculado com a ajuda do módulo *Flow Monitor* disponível no NS-3 [CFR09]. Esse módulo armazena um atributo chamado *timeLastRxPacket*, definido como o tempo absoluto em que o último pacote de um fluxo foi recebido, ou seja, o tempo em que um fluxo para de receber dados. Vale ressaltar que há uma diferença no funcionamento do nosso módulo se comparado com as aplicações que usamos para simular os fluxos TCP. O par de aplicações *BulkSend* e *PacketSink* não funciona em um modelo cliente-servidor comum. Ou seja, um nó *PacketSink* não inicia uma conexão com um nó *BulkSend* para começar a receber os dados. Ao invés disso, uma aplicação *BulkSend* inicia uma conexão com um nó *PacketSink* e, quando a conexão estiver estabelecida, começa a enviar dados. Nossas implementações de cliente e servidor do QUIC, porém, funcionam da forma tradicional: um cliente inicia uma conexão com um servidor e faz uma requisição por uma quantidade determinada de bytes. O servidor recebe a requisição e envia a quantidade de bytes requerida. Nós argumentamos que, apesar dessa diferença de funcionamento, para calcular o tempo total de transmissão de dados basta armazenar o valor do atributo *timeLastRxPacket*. No caso de um fluxo TCP, se iniciarmos a aplicação *BulkSend* no tempo zero de simulação, o atributo *timeLastRxPacket* no fluxo de um nó *BulkSend* para um *PacketSink* indicará exatamente o tempo total de transmissão dos dados, incluindo o *3-way handshake*. Da mesma forma, se iniciarmos um cliente QUIC no tempo zero de simulação, o tempo total de transmissão de dados será o valor armazenado no atributo *timeLastRxPacket* no fluxo de um servidor para um cliente QUIC.

Calculamos a justiça da seguinte forma, conforme definido por [Jai90]. Dado um conjunto de taxas de recebimento $(x_1, x_2, \ldots, x_n)$, o **índice de justiça** do conjunto pode ser escrito como:

$$f(x_1, x_2, \ldots, x_n) = \frac{(\sum_{i=1}^{n} x_i)^2}{n * \sum_{i=1}^{n} x_i^2}$$

Supondo que as taxas de recebimento serão maiores que zero, esse índice sempre estará entre 0 e 1. Se todos os nós tiverem taxa de recebimento igual, o índice será igual a 1. Portanto, quanto mais próximo de 1, mais justo será o uso da largura de banda entre os fluxos. Neste trabalho, calculamos a justiça entre fluxos QUIC e entre um fluxo QUIC e um fluxo TCP. O *Flow Monitor* também armazena atributos chamados *timeFirstRxPacket* e *rxBytes*, definidos como o valor absoluto em que o primeiro pacote em um fluxo foi recebido, ou seja, o tempo em que um fluxo começa a receber dados e a quantidade de dados recebida no fluxo, respectivamente. Dessa forma, segundo [CFR09], podemos calcular a taxa média de recebimento de bits ($bits/s$) como:

$$\overline{B_{rx}} = \frac{8 * rxBytes}{timeLastRxPacket - timeFirstRxPacket}$$

A maior parte dos resultados foram obtidos com 1 execução de cada configuração de cenário. Por esses resultados serem provenientes de simulações com caráter determinístico e pelo tempo total de transmissão calculado ser o de simulação e não o tempo de relógio real, não é necessário rodar as simulações mais de uma vez. A exceção fica por conta dos experimentos que possuem taxa de perda de pacotes. Nesses casos, rodamos 30 vezes cada experimento e calculamos as respectivas médias e desvios padrão.

Por fim, é importante notar que o código que integramos do QUIC é o disponibilizado pela Google como uma biblioteca *standalone*. Esse código, segundo os próprios desenvolvedores não foi escrito para rodar em produção [pla]. Outros trabalhos, como [BG16] e [CDCM15], reportaram como limitações de suas medições justamente esse ponto. Em [KJC+17], porém, os autores calibram esse código para conseguir paridade de desempenho com os servidores da Google. Eles reportam que aumentaram o valor máximo do tamanho da janela de congestionamento e que arrumaram um erro no algoritmo de partida lenta. Essas modificações já estão incorporadas no código que integramos. Os autores também argumentam que a implementação de servidor não teria bom desempenho em produção por não conseguir atender requisições simultâneas. Nas nossas simulações, assim como nas medições que eles fizeram, há apenas um cliente acessando o servidor por vez. Dessa forma, acreditamos que nosso módulo terá desempenho similar ao observado pelas medições.

5.1 Resultados

A Figura 5.3 apresenta o tempo total de transmissão dos dados, na topologia simples, ao mantermos fixos o atraso e a largura de banda, ao variarmos a quantidade de dados enviada e desconsiderarmos perdas de pacotes pelo meio físico. O gráfico está em escala logarítmica por conta da grande diferença entre o tempo total de transmissão de 10KB e 210MB. O QUIC tem desempenho melhor do que todas as variações do TCP. Por conta da escala, a impressão visual que o gráfico passa é que a diferença no tempo de transmissão é maior para quantidades pequenas de dados e que essa diferença diminui conforme a quantidade de dados transmitida aumenta. Na verdade, entretanto, a diferença aumenta. Para 10KB, a diferença entre o QUIC e os TCPs não passa de alguns centésimos de segundo. Já para 210MB, a diferença passa a ser de mais de 10 segundos. Note que, com tantos protocolos, muitos deles acabam possuindo o mesmo comportamento, formando 4 agregados de curvas. Para facilitar o entendimento, seguem as curvas que estão próximas entre si no gráfico, do melhor para o pior resultado:

- Agregado 1: QUIC;

- Agregado 2: TCP Hybla;
- Agregado 3: TCP NewReno, TCP BIC, TCP CUBIC, TCP HighSpeed, TCP Htcp, TCP Illinois, TCP Ledbat, TCP Scalable, TCP Veno, TCP Westwood, TCP Yeah e,
- Agregado 4: TcpVegas

Como muitas variações do TCP tiveram desempenho similar umas às outras em todas as outras simulações, e o TCP CUBIC foi aquele que teve o resultado mais próximo do QUIC, daqui em diante somente apresentaremos os resultados das simulações do TCP CUBIC para comparação. Além do mais, o CUBIC também é o algoritmo usado pelo QUIC. A razão pela qual o QUIC tem desempenho melhor é porque sua implementação do CUBIC incrementa a janela de congestionamento mais agressivamente e mantém a utilização do enlace mais próxima de sua capacidade máxima. Por exemplo, para o caso de 210MB transmitidos, o QUIC alcança taxa média de recebimento de 9,96Mbps enquanto o TCP alcança 9,89Mbps. Isso corresponde a 99% e 98% da largura de banda do enlace, respectivamente.

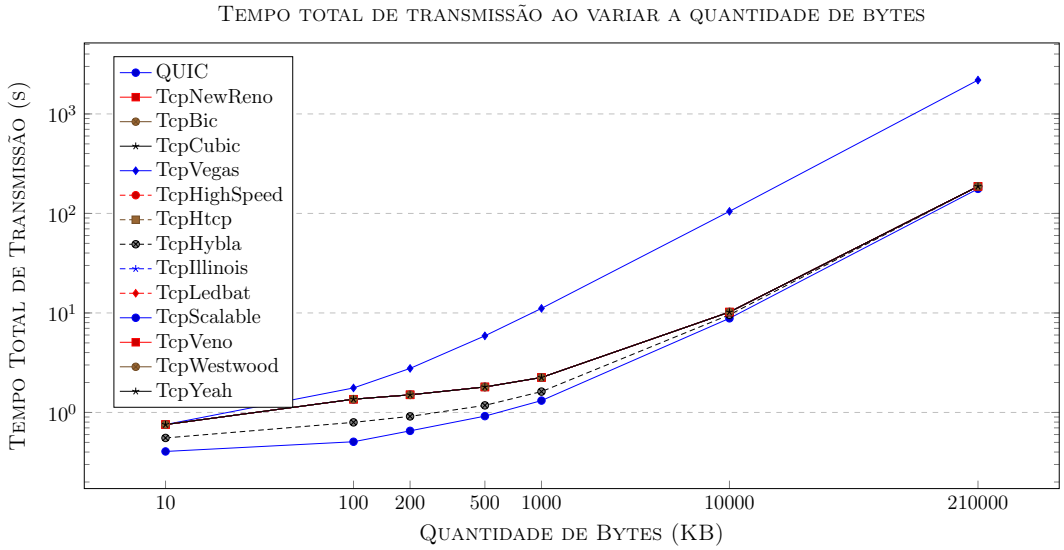

Figura 5.3: *Comparação do tempo total de transmissão entre o QUIC e os vários TCP ao variarmos a quantidade de dados enviados no cenário simples (sem perda de pacotes).*

Na Figura 5.4, apresentamos o tempo total de transmissão de dados para a topologia simples, ao variarmos a largura de banda, desconsiderarmos perdas de pacotes e fixarmos a quantidade de dados enviada e o tempo de propagação do enlace. Mais uma vez, o QUIC tem desempenho melhor que o TCP em todos os casos. O fato de o QUIC incrementar mais agressivamente a janela de congestionamento é comprovado pela maior diferença no tempo de transmissão entre os protocolos conforme a largura de banda cresce. Para o caso de 5Mbps, o QUIC possui taxa média de recebimento de 4,17Mbps e o TCP de 2,87Mbps. Já para o caso de 1Gbps, esse valor é de 11,07Mbps para o QUIC e de 3,99Mbps para o TCP. Isso indica que o TCP não consegue incrementar suficientemente sua janela de congestionamento antes da transferência de dados terminar, utilizando pouco da largura de banda disponível. O QUIC, por sua vez, consegue utilizar quase 3 vezes mais dos recursos disponíveis. Isso impacta diretamente no tempo total de transmissão dos dados. Não por acaso o QUIC é quase 3 vezes mais rápido para completar a transmissão.

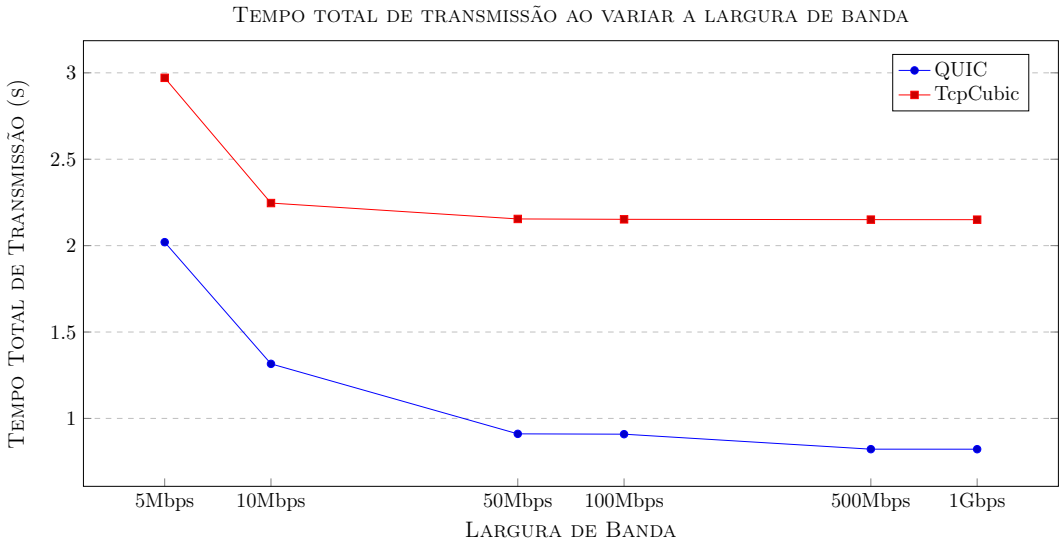

Figura 5.4: *Comparação do tempo total de transmissão entre o QUIC e o TCP CUBIC ao variarmos a largura de banda no cenário simples (sem perda de pacotes).*

Já na Figura 5.5, calculamos o tempo total de transmissão dos dados no cenário simples, sem considerar perdas de pacotes pelo meio físico, fixando a largura de banda e a quantidade de dados enviada e variando o atraso no enlace entre os dois nós. Podemos perceber que, como esperado, o tempo total de transmissão aumenta conforme o atraso aumenta. Pelo gráfico, podemos ver que a diferença de desempenho entre os protocolos também aumenta com atrasos maiores. Mais uma vez, o QUIC é capaz de estimar e utilizar melhor os recursos disponíveis da rede. Para atrasos de 10ms, em média, o QUIC consegue utilizar 90% da largura de banda disponível e o TCP 78%. Já para atrasos de 200ms, esses valores são de 23% e 10%, respectivamente. Essa diferença na utilização da largura de banda (o QUIC mantém a taxa média de recebimento pouco mais de 2 vezes maior que a do TCP) é consistente com a diferença no tempo total de transmissão: o QUIC é pouco mais de 2 vezes mais rápido que o TCP para concluir a transmissão no caso de 200ms de atraso no enlace.

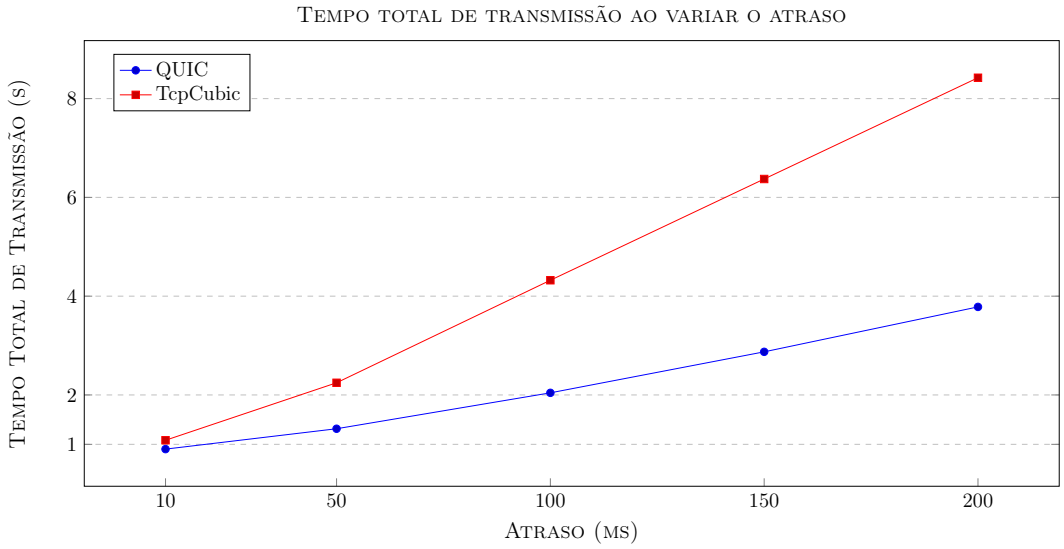

Figura 5.5: *Comparação do tempo total de transmissão entre o QUIC e o TCP CUBIC ao variarmos o atraso no cenário simples (sem perda de pacotes).*

Os resultados do último cenário usando a topologia simples são mostrados na Figura 5.6. Nele, fixamos a quantidade de dados enviada, o atraso e a largura de banda e variamos a taxa de erros pelo meio físico. Para cada taxa de erro, rodamos 30 vezes os experimentos. O gráfico apresenta o tempo total de transmissão médio e barras de erro que representam a média somada com e subtraída do desvio padrão. Omitimos os resultados com 10% de erro porque o desempenho do TCP CUBIC degrada significativamente, extrapolando a escala do gráfico. Podemos ver que o QUIC tem desempenho melhor e tem pouquíssima variabilidade para taxas de erro iguais a 0,01% e 0,1% (os desvios padrão para esses 2 casos ficam abaixo de 0,1, dando a impressão que as barras de erro não estão impressas no gráfico). O TCP CUBIC, por sua vez, apresenta grande degradação no desempenho e alta variabilidade nos resultados. Apesar de os dois protocolos identificarem congestionamento por perdas de pacote, o QUIC reage mais rápido do que o TCP quando percebe que há banda disponível para uso. Quando a taxa de erros é alta, o TCP opera com uma taxa de transmissão muito baixa, impactando o tempo total de transmissão dos dados. Em particular, seu desempenho é pior quando os erros acontecem em intervalos curtos e regulares.

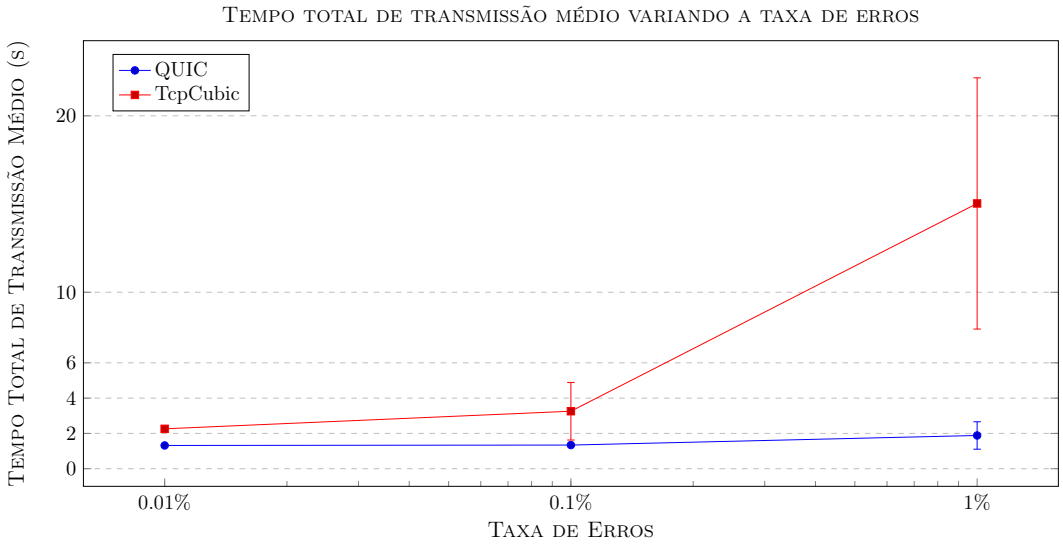

Figura 5.6: *Comparação do tempo total de transmissão entre o QUIC e o TCP CUBIC ao variarmos a taxa de erros no cenário simples.*

O gráfico da Figura 5.7 apresenta os resultados das simulações, na topologia Dumbbell, ao fixarmos a largura de banda e o tempo de propagação do enlace entre os roteadores, sem considerar perdas por falhas físicas e variando a quantidade de dados enviada. O gráfico mostra o tempo total de transmissão do fluxo mais lento. Esse padrão também é adotado para os gráficos das Figuras 5.8 e 5.9. Para preservar a escala do gráfico, omitimos os resultados para transmissões de 210MB. O QUIC tem desempenho melhor que o TCP para todos os valores. Assim como no caso da topologia simples, o QUIC consegue incrementar mais rapidamente sua janela de congestionamento e utilizar melhor os recursos da rede disponíveis, fazendo com que esse protocolo consiga transmitir dados mais rápido que o TCP.

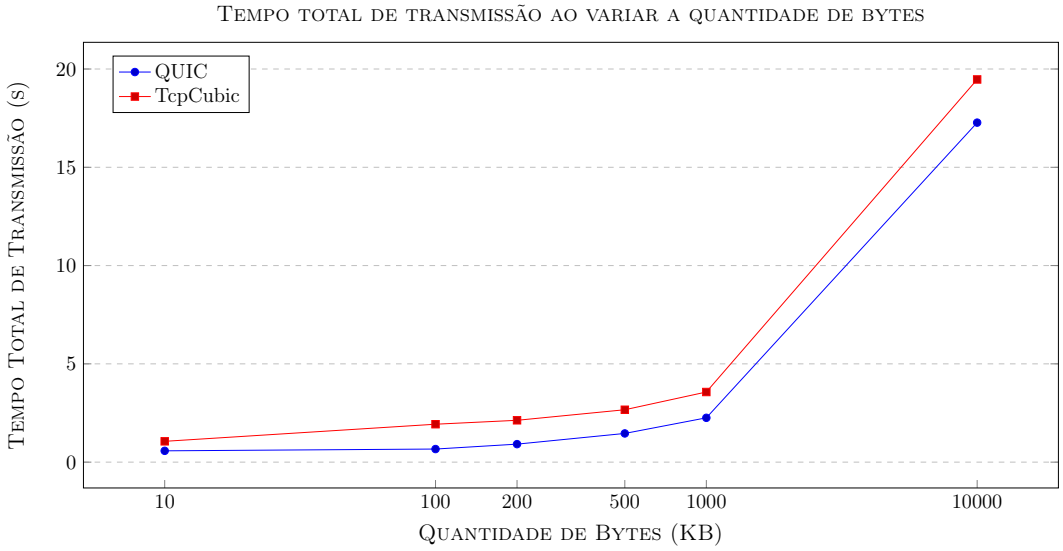

Figura 5.7: *Comparação do tempo total de transmissão entre o QUIC e o TCP CUBIC ao variarmos a quantidade de dados enviados na topologia Dumbbell, desconsiderando falhas pelo meio físico.*

A Figura 5.8 mostra como o tempo total de transmissão se comporta, na topologia Dumbbell, ao mantermos fixos o atraso e a quantidade de dados enviada, desconsiderarmos perdas por falhas físicas e variarmos a largura de banda do enlace entre os roteadores. Assim como na topologia simples, o QUIC tem desempenho melhor que o TCP para todos os valores testados. Note que, a

partir de 50Mbps, o tempo total de transmissão é pouco afetado pelo aumento da largura de banda para ambos os protocolos. Novamente, a melhor utilização dos recursos da rede e o incremento mais agressivo da janela de congestionamento faz com que o QUIC consiga tempos menores de transmissão que o TCP.

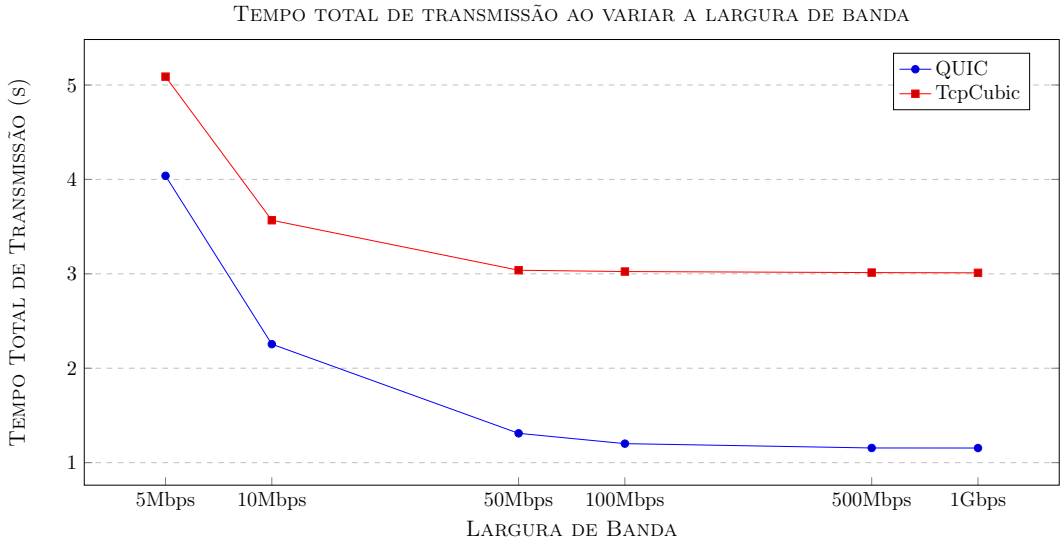

Figura 5.8: *Comparação do tempo total de transmissão entre o QUIC e o TCP CUBIC ao variarmos a largura de banda na topologia Dumbbell, sem considerar perdas pelo meio físico.*

Na figura 5.9, apresentamos os resultados das simulações para a topologia Dumbbell ao variarmos o tempo de propagação no enlace entre os roteadores, mantendo fixas a quantidade de dados enviada e a largura de banda e sem considerar perdas de pacotes pelo meio físico. Como esperado, o tempo total de transmissão aumenta conforme o RTT do caminho entre os nós também aumenta. Note que o gráfico é praticamente igual ao da Figura 5.5, na qual usamos as mesmas configurações para a topologia simples. Na topologia Dumbbell o QUIC também tem desempenho melhor que o TCP por sua capacidade de estimar e utilizar melhor a banda disponível e por se recuperar com mais eficiência quando pacotes são perdidos.

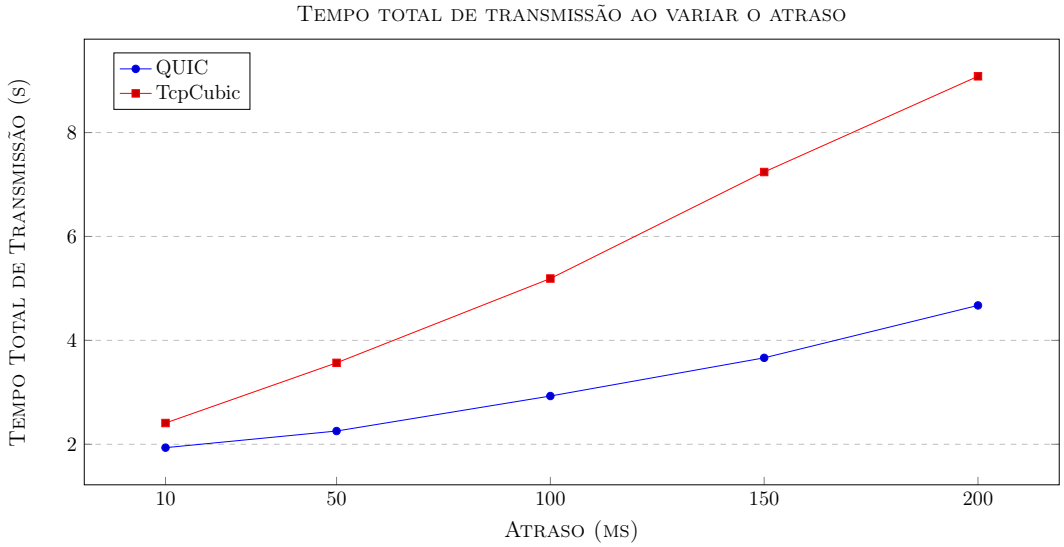

Figura 5.9: *Comparação do tempo total de transmissão entre o QUIC e o TCP CUBIC ao variarmos o atraso no cenário Dumbbell, desconsiderando perdas de pacotes pelo meio físico.*

Já a Figura 5.10 compara o desempenho do QUIC e do TCP CUBIC, para a topologia Dumbbell, ao mantermos fixos o atraso, a quantidade de dados enviada e a largura de banda, mas adicionando uma taxa de erros no enlace entre os roteadores. Rodamos as simulações 30 vezes para cada valor de taxa de erro e apresentamos a média do tempo total de transmissão junto com barras de erro que indicam a média acrescida e decrescida do desvio padrão. É notável que o QUIC tem pouca variação na transmissão de dados na presença de mais falhas no meio físico sendo que, para taxas de erro iguais a 0,01% e 0,1%, a oscilação é tão pequena que no gráfico não é possível ver as barras de erro. O TCP, por sua vez, apresenta mais variabilidade nos resultados e seu desempenho degrada significativamente conforme a taxa de erros aumenta. Isso extrapola a escala do gráfico para 10% de taxa de erro. Por isso, omitimos os resultados para esse valor. O melhor desempenho do QUIC pode ser explicado pelo seu algoritmo mais eficiente de recuperação ao detectar perdas de pacotes. Isso faz com que ele consiga utilizar mais da banda disponível. O TCP, entretanto, não incrementa sua janela de congestionamento tão rapidamente. Quando perdas pelo meio físico são mais frequentes, ele infere que a rede está congestionada e mantém baixa sua taxa de transmissão.

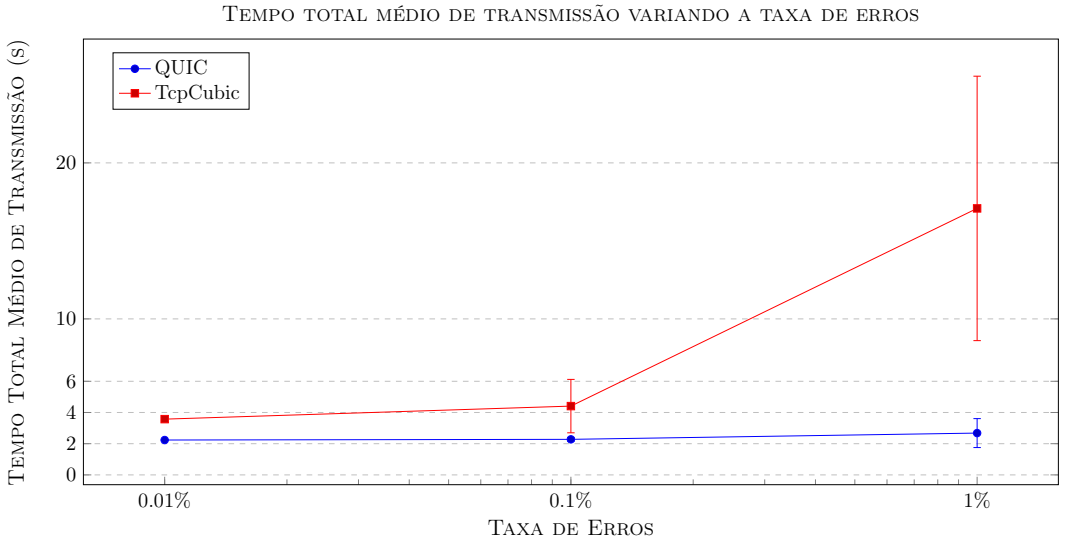

Figura 5.10: *Comparação do tempo total de transmissão entre o QUIC e o TCP CUBIC ao variarmos a taxa de erro no cenário Dumbbell.*

Para ilustrar como em simulações é mais fácil escalar os experimentos, a Figura 5.11 apresenta os resultados de desempenho, para a topologia Dumbbell, ao mantermos fixos o atraso, largura de banda e quantidade de dados enviada, desconsiderando perdas de pacotes pelo meio físico e variando o número de nós da topologia. Por exemplo, no gráfico, o valor de 20 nós indica que no experimento tínhamos 10 nós conectados em cada roteador. Ou seja, 10 nós enviando e 10 recebendo dados. O gráfico mostra a mediana do tempo total de transmissão entre todos os fluxos e também marca o valor máximo e mínimo observado por meio de barras de erro. Para 4 nós, mostramos apenas o tempo total de transmissão para o fluxo mais lento. Vemos que o QUIC possui desempenho melhor para todos os números de nós porque, em todos os casos, o QUIC apresentou o fluxo mais rápido e o TCP o fluxo mais lento. Além disso, a mediana do QUIC também sempre fica abaixo da mediana do TCP. Considere o caso de 80 nós de cada lado da topologia. Vemos que, apesar de o fluxo TCP mais rápido estar próximo do fluxo QUIC mais rápido, o fluxo TCP mais lento está distante da mediana. Isso quer dizer que pelo menos 1 fluxo TCP demorou quase 6 segundos a mais que a maioria dos fluxos. Já o fluxo QUIC mais lento está próximo da mediana, sendo apenas 3 segundos mais lento que a maioria. Outro caso interessante é o de 20 nós enviando e recebendo dados. Nesse caso, a mediana do TCP está praticamente igual ao mínimo. Isso quer dizer que pelo menos metade dos fluxos TCP tiveram desempenho próximo ao fluxo mais rápido. Já para o QUIC acontece o contrário. Sua mediana está praticamente igual ao máximo, o que quer dizer que pelo menos metade

dos fluxos QUIC tiveram desempenho próximo ao fluxo mais devagar.

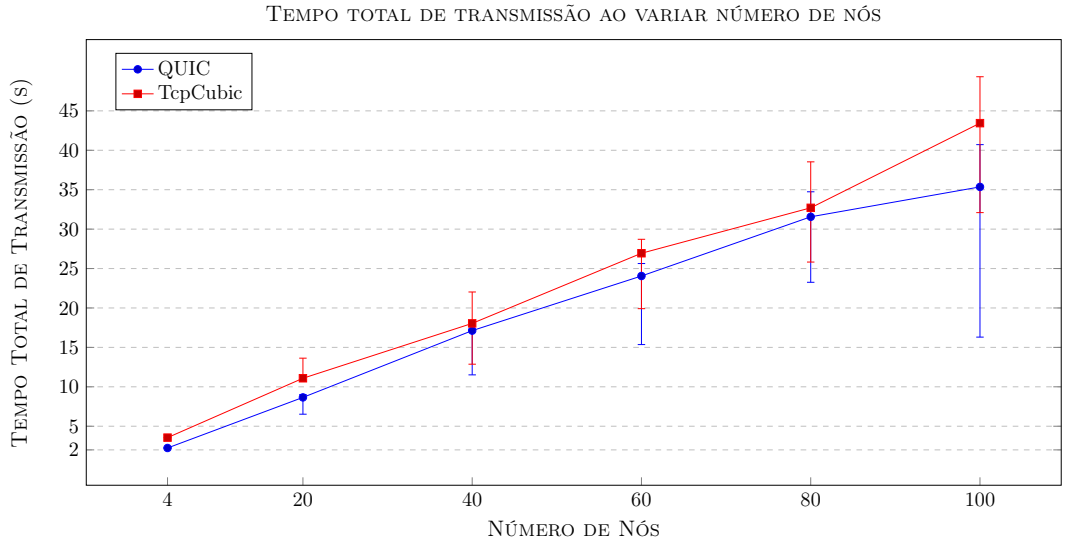

Figura 5.11: *Comparação do tempo total de transmissão entre o QUIC e o TCP CUBIC ao variarmos o número de nós no cenário Dumbbell, sem considerar perdas de pacotes pelo meio físico.*

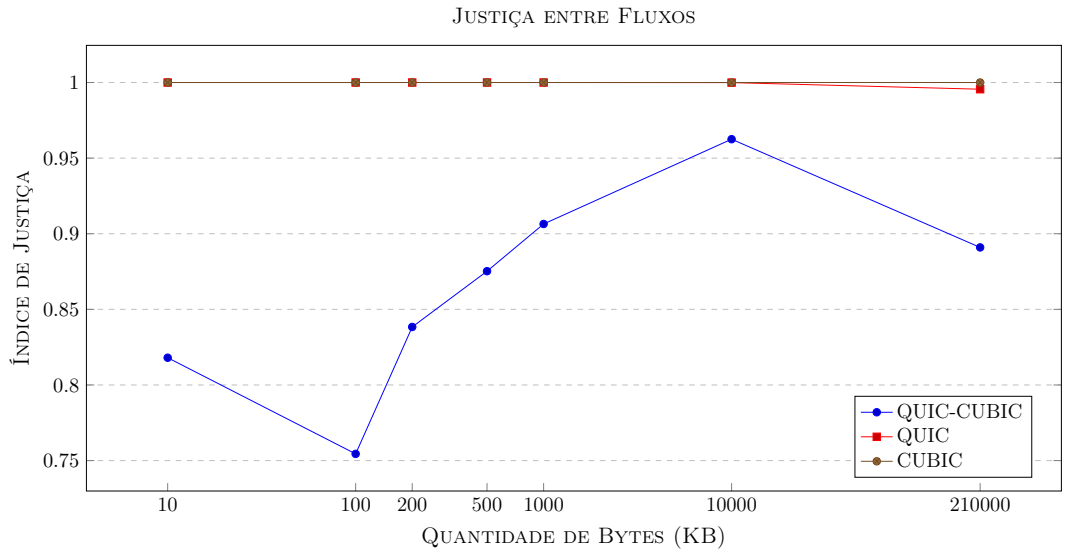

Figura 5.12: *Índice de justiça conforme a variação da quantidade de bytes transmitida.*

Por fim, analisamos a justiça entre o QUIC e TCP CUBIC. Lembramos que essa propriedade é importante para qualquer protocolo de transporte porque, sem ela, ocorrerá degradação no desempenho em fluxos concorrentes. O gráfico da Figura 5.12 mostra o valor do índice de justiça para diferentes quantidades de dados transmitidas. Realizamos os experimentos na topologia Dumbbell, sem perdas de pacote pelo meio físico, em três situações: uma com 2 fluxos QUIC, outra com 2 fluxos TCP Cubic e a última com 1 fluxo TCP CUBIC e 1 fluxo QUIC concorrentes. Como esperado, o índice de justiça para as duas primeiras situações foi próximo de 1, o que significa que ambos os fluxos compartilharam de forma justa os recursos da rede. Porém, para 1 fluxo TCP CUBIC competindo com 1 fluxo QUIC a justiça não se repete, apesar de o QUIC usar o CUBIC como algoritmo de controle de congestionamento. O motivo para esse comportamento é que o QUIC incrementa sua janela de congestionamento de forma mais agressiva, tanto em termos de frequência quanto de tamanho do incremento. Por isso, o QUIC consegue adquirir a banda disponível mais rápido que o

TCP. Essa característica foi mais acentuada na transmissão de 100KB, na qual o QUIC alcançou uma taxa média de recebimento 3,6 vezes maior que a do TCP.

5.2 Validação

Separamos em duas partes a validação deste trabalho: a validação do funcionamento do módulo e a validação dos resultados de simulação.

Entendemos por validação do funcionamento do módulo o fluxo correto de trocas de mensagens entre cliente e servidor e a transferência completa de dados. Ou seja, verificamos que o processo de estabelecimento de conexão, transferência dos dados e encerramento da conexão segue o fluxo esperado e que todos os dados enviados pelo servidor de fato chegam no cliente, respectivamente. Verificamos a comunicação entre cliente e servidor por meio de mensagens de *log* no código do módulo que nos permitia acompanhar a troca de mensagens entre as duas pontas. Para garantir que todos os dados que o servidor envia chegam no cliente, geramos uma sequência aleatória de caracteres no servidor e verificamos que a sequência que o cliente recebe é igual a que construímos no servidor.

Para validar os resultados de simulação, comparamos os resultados que obtivemos com os resultados de medição apresentados em [KJC+17]. Alguns dos resultados que os autores desse trabalho apresentam não se aplicam aos nossos cenários de simulação como, por exemplo, nas medições de qualidade de experiência para *streaming* de vídeos. Por isso, a seguir citamos apenas aqueles que fazem sentido no contexto dos nossos experimentos.

1. Para ambientes *desktop*, o desempenho do QUIC é melhor que o do TCP CUBIC em quase todos os cenários e,

2. O QUIC é justo com outros fluxos QUIC. Porém, ao competir com fluxos TCP CUBIC, o QUIC é injusto, consumindo mais do dobro da largura de banda apropriada.

Nossos resultados são consistentes com ambos os resultados citados acima. Em todos nossos cenários de simulação, o QUIC conseguiu desempenho melhor que o TCP CUBIC e apresentou os mesmos problemas de justiça com fluxos TCP CUBIC concorrentes. É importante ressaltar que, em [KJC+17], os autores reportaram que o TCP CUBIC obteve desempenho melhor que o QUIC em alguns cenários específicos: em casos em que há alta reordenação de pacotes e em casos em que o cliente faz *download* de um grande número de objetos pequenos. Porém, esses dois cenários estão fora do escopo deste trabalho e são apontados como posíveis trabalhos futuros.

Capítulo 6

Conclusão

O protocolo QUIC é um novo método de transporte de dados em redes de computadores desenvolvido pela Google. Esse protocolo foi construído usando como base pesquisas em protocolos de transporte e tem como objetivo aplicar e testar rapidamente novas técnicas para conseguir desempenho melhor que o TCP, o protocolo de transporte mais utilizado na Internet atualmente. O QUIC está em processo de padronização pela IETF e todas as requisições feitas pelo *Chrome* para serviços da Google já são transportadas por esse protocolo, fazendo com que ele já represente parte significativa do tráfego total de dados na Internet. A Google publicou relatórios de experimentos que fizeram com QUIC alegando que ele possui desempenho melhor que o TCP em diversas situações. Porém, esses experimentos não possuem informações suficientes para serem reproduzidos e possuir a infraestrutura de testes que a Google utilizou para realizar as medições não é plausível. Com a discussão sobre se vale a pena ou não substituir o TCP pelo QUIC, uma maneira para se fazer análises de desempenho cujos resultados sejam reprodutíveis e verificáveis é necessária para se tomar uma decisão informada sobre esse assunto.

Neste trabalho, estudamos as 3 técnicas de análise de desempenho: modelagem analítica, simulação e medição. Com essa base e porque, até onde sabemos, não existe um outro trabalho que avalia o desempenho do QUIC por meio de simulações, decidimos usar essa técnica para a nossa análise. Também pesquisamos na literatura como são feitas análises de desempenho de protocolos de transporte, especificamente do TCP e do QUIC, e reunimos informações relevantes para essas análises: qual técnica é utilizada, quais topologias e configurações de parâmetros de rede são normalmente escolhidas, quais métricas são geralmente calculadas e como os resultados são validados. Escrevemos um módulo que implementa o protocolo QUIC no simulador de redes NS-3 e fizemos experimentos em diversos cenários de rede. Avaliamos o protocolo QUIC em termos do tempo total de transmissão de dados e do índice de justiça entre fluxos, comparando seu desempenho principalmente com o TCP CUBIC. Dessa forma, atingimos todos os objetivos listados na seção 1.2.

Nossa principal conclusão desta análise foi que o QUIC apresenta desempenho melhor que o TCP CUBIC, em termos do tempo total de transmissão de dados, em todos os cenários de rede simulados. Além disso, também concluímos que o QUIC apresenta justiça entre fluxos QUIC mas que um fluxo QUIC não é justo com um fluxo TCP CUBIC. Esses resultados foram validados e estão em conformidade com os resultados de medição apresentados por [KJC+17].

6.1 Contribuições

Como citado na subseção 1.3, as contribuições deste trabalho são a análise inédita de desempenho do protocolo QUIC por meio de simulações, que também serve para validação de resultados de medição ou de modelagem analítica e a implementação de um módulo do QUIC para o NS-3. Por fim, publicamos como foi feita essa implementação e apresentamos resultados preliminares no artigo **Y. Couto, D. Camarinha, D. M. Batista, "nsQUIC: Uma Extensão para Simulação do Protocolo QUIC no NS-3". In: Salão de Ferramentas do SBRC 2018, 2018. Anais do Salão de Ferramentas do SBRC 2018, 2018**.

6.2 Limitações

As principais limitações deste trabalho são:

1. O módulo do QUIC para o NS-3 ainda não dá suporte para estabelecimento de conexão em 0-RTT. Em [KJC+17], os autores relatam que a funcionalidade de 0-RTT do QUIC acarreta em ganhos significativos de desempenho, principalmente em transferências de pequenas quantidades de dados. Apesar disso, essa funcionalidade não deve mudar nossos resultados porque o QUIC já apresentou desempenho melhor em todos os cenários de rede simulados e,

2. A implementação do CUBIC no NS-3 é baseada no artigo do CUBIC e em uma implementação antiga do CUBIC no kernel do Linux. Desde então, a própria Google descobriu e corrigiu um erro no algoritmo que impactava diretamente o desempenho do CUBIC em certas situações. É possível que a versão mais atualizada do CUBIC tenha desempenho melhor que o QUIC em algum cenário de rede.

6.3 Trabalhos Futuros

Para aprimorar a análise de desempenho do QUIC, muito trabalho ainda pode ser feito:

- Utilizar o módulo para calcular métricas de desempenho que ainda não foram calculadas em outros trabalhos relacionados ao desempenho do QUIC como a capacidade de utilização dos enlaces e o *goodput*, por exemplo;

- Habilitar a funcionalidade de 0-RTT no módulo do QUIC e verificar se ganhos de desempenho similares aos reportados por [KJC+17] são observados;

- Atualizar a implementação do CUBIC no NS-3 com novas alterações e correções de erro existentes no kernel do Linux e rodar as simulações novamente para verificar se os resultados mudam;

- Implementar o BBR no NS-3 e rodar as simulações para comparar seu desempenho com o do TCP CUBIC e o do QUIC;

- Em [KJC+17], os autores relatam que, em situações em que acontecem reordenação de pacotes, o desempenho do QUIC é pior que o do TCP CUBIC em diversos cenários. Criar uma simulação que apresente essa situação é importante para que propostas de soluções para esse problema possam verificar que o desempenho do protocolo de fato melhora;

- Outro resultado de [KJC+17] é que o QUIC tem desempenho ruim no tempo total de transmissão em cenários nos quais um cliente faz o *download* de um grande número de objetos pequenos. Pelo mesmo motivo que no item anterior, é importante criar um cenário que simule essa situação;

- Incluir mais cenários de simulação em redes sem fio e em redes mistas (uma parte sem fio e uma parte cabeada) para avaliar o desempenho do QUIC nesses ambientes de rede e,

- Atualizar o módulo do QUIC quando a RFC do protocolo estiver publicada.

Apêndice A

Outras Publicações

Além da publicação já citada sobre o módulo do QUIC no salão de ferramentas do SBRC, durante o período em que fui aluno de mestrado, também participei das seguintes publicações:

- **R. Siqueira, D. Camarinha, M. Wen, P. Meirelles and F. Kon, "Continuous Delivery: Building Trust in a Large-Scale, Complex Government Organization", in IEEE Software, vol. 35, no. 2, pp. 38-43, March/April 2018.** Artigo sobre as experiências que tivemos com entrega contínua de sistemas durante 2 anos de trabalho com o governo federal brasileiro, no contexto do projeto Portal do Software Público Brasileiro.
- **D. Guedes, P. Meirelles, R. Manzo, and D. Camarinha, "Mezuro: Understanding source code metrics". In: The 13th International Conference on Open Source Systems, 2017, Buenos Aires. Poster Proceedings of the 13th International Conference on Open Source Systems, 2017. p. 15-18.** Pôster sobre a plataforma de análise estática de código fonte, o Mezuro, desenvolvida no Centro de Competência em Software Livre do IME e que é um dos sistemas que compõem o Portal do Software Público Brasileiro.

Referências Bibliográficas

[ASS+16] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J Alex Halderman, Viktor Dukhovni et al. Drown: Breaking tls using sslv2. Em *USENIX Security Symposium*, páginas 689–706, 2016. 17

[ATRK10] Alexander Afanasyev, Neil Tilley, Peter Reiher e Leonard Kleinrock. Host-to-host congestion control for tcp. *IEEE Communications surveys & tutorials*, 12(3):304–342, 2010. 12

[BBR] TCP BBR congestion control comes to GCP - your Internet just got faster. https://cloudplatform.googleblog.com/2017/07/TCP-BBR-congestion-control-comes-to-GCP-your-Internet-just-got-faster.html. Acessado em: 17/09/2018. 13

[BG16] Prasenjeet Biswal e Omprakash Gnawali. Does quic make the web faster? Em *Global Communications Conference (GLOBECOM), 2016 IEEE*, páginas 1–6. IEEE, 2016. 30, 32, 39

[Bir11] Ian Bird. Computing for the large hadron collider. *Annual Review of Nuclear and Particle Science*, 61:99–118, 2011. 1

[Bra94] Robert T. Braden. T/TCP – TCP Extensions for Transactions Functional Specification. RFC 1644, Julho 1994. 9

[CCG+16] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh e Van Jacobson. Bbr: Congestion-based congestion control. *Queue*, 14(5):50, 2016. 13

[CDCM15] Gaetano Carlucci, Luca De Cicco e Saverio Mascolo. Http over udp: an experimental investigation of quic. Em *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, páginas 609–614. ACM, 2015. 21, 30, 32, 39

[CFR09] Gustavo Carneiro, Pedro Fortuna e Manuel Ricardo. Flowmonitor: a network monitoring framework for the network simulator 3 (ns-3). Em *Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*, página 1. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009. 38, 39

[CGM+02] Claudio Casetti, Mario Gerla, Saverio Mascolo, Medy Y Sanadidi e Ren Wang. Tcp westwood: end-to-end congestion control for wired/wireless networks. *Wireless Networks*, 8(5):467–479, 2002. 27, 29, 30

[DCCM13] Nandita Dukkipati, Neal Cardwell, Yuchung Cheng e Matt Mathis. Tail Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Losses. Internet-Draft draft-dukkipati-tcpm-tcp-loss-probe-01, Internet Engineering Task Force, Fevereiro 2013. Work in Progress. 21

[DR06] Tim Dierks e Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346, Abril 2006. 10

[DVdVKI99] Marco De Vivo, Gabriela O de Vivo, Roberto Koeneke e Germinal Isern. Internet vulnerabilities related to tcp/ip and t/tcp. *ACM SIGCOMM Computer Communication Review*, 29(1):81–85, 1999. 9

[FMMR13] Sally Floyd, Jamshid Mahdavi, Matt Mathis e Dr. Allyn Romanow. TCP Selective Acknowledgment Options. RFC 2018, Março 2013. 23

[GN] GN Build Configuration. https://www.chromium.org/developers/gn-build-configuration. Acessado em: 17/09/2018. 34

[GNUS13] Siddharth Gangadhar, Truc Anh N Nguyen, Greeshma Umapathi e James PG Sterbenz. Tcp westwood (+) protocol implementation in ns-3. Em *Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques*, páginas 167–175. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2013. 27, 29, 30

[HBZ17] Mario Hock, Roland Bless e Martina Zitterbart. Experimental evaluation of bbr congestion control. Em *Network Protocols (ICNP), 2017 IEEE 25th International Conference on*, páginas 1–10. IEEE, 2017. 28, 29, 30

[HRX08] Sangtae Ha, Injong Rhee e Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS Operating Systems Review*, 42(5):64–74, 2008. 12, 26, 29, 30

[Ind17] Cisco Visual Networking Index. The zettabyte era–trends and analysis. *Cisco white paper*, 2017. 1

[IWS] Internet world stats - usage and population statistics. http://www.internetworldstats.com/stats.htm. Acessado em: 17/09/2018. 1

[Iye] Jana Iyengar. Quic. redefining internet transport. https://docs.google.com/presentation/d/15e1bLKYeN56GL1oTJSF9OZiUsI-rcxisLo9dEyDkWQs/edit?usp=sharing. Acessado em: 12/04/2016. ix, 14

[Jac88] Van Jacobson. Congestion avoidance and control. Em *ACM SIGCOMM computer communication review*, volume 18, páginas 314–329. ACM, 1988. 11

[Jai90] Raj Jain. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. John Wiley & Sons, 1990. xi, 5, 6, 29, 32, 38

[JRT02] Andreas Jungmaier, Eric Rescorla e Michael Tuexen. Transport layer security over stream control transmission protocol. *IETF, Standards Track RFC*, 3436, 2002. 23

[Kau05] Charlie Kaufman. Internet Key Exchange (IKEv2) Protocol. RFC 4306, Dezembro 2005. 19

[KJC+17] Arash Molavi Kakhki, Samuel Jero, David Choffnes, Cristina Nita-Rotaru e Alan Mislove. Taking a Long Look at QUIC: An Approach for Rigorous Evaluation of Rapidly Evolving Transport Protocols. Em *Proceedings of the 2017 Internet Measurement Conference*, IMC '17, páginas 290–303. ACM, 2017. 7, 29, 31, 32, 37, 39, 47, 49, 50

[KK16] J Kurose e W Keith. *Computer networking: a top down approach, 7th Edition*. Pearson, 2016. 10, 11

[KP87] Phil Karn e Craig Partridge. Improving round-trip time estimates in reliable transport protocols. Em *ACM SIGCOMM Computer Communication Review*, volume 17, páginas 2–7. ACM, 1987. 22

[Lan] Wan-Teh Langley, Adam e Chang. QUIC Crypto. https://docs.google.com/document/d/1g5nIXAIkN_Y-7XJW5K45IblHd_L2f5LTaDUDwvZ5L6g/edit. Acessado em: 21/07/2016. ix, 18, 19

[LCK14] Brett Levasseur, Mark Claypool e Robert Kinicki. A tcp cubic implementation in ns-3. Em *Proceedings of the 2014 Workshop on ns-3*, página 3. ACM, 2014. 26, 29, 30

[LHC] The large hadron collider. https://home.cern/topics/large-hadron-collider. Acessado em: 17/09/2018. 1

[LIGa] Gravitational waves detected 100 years after einstein's prediction. https://www.ligo.caltech.edu/news/ligo20160211. Acessado em: 17/09/2018. 1

[LIGb] Laser interferometer gravitational-wave observatory. https://www.ligo.caltech.edu/. Acessado em: 17/09/2018. 1

[LIGc] LIGO Technology. https://www.ligo.caltech.edu/page/ligo-technology. Acessado em: 17/09/2018. 1

[LLS07] Yee-Ting Li, Douglas Leith e Robert N Shorten. Experimental evaluation of tcp protocols for high-speed networks. *IEEE/ACM Transactions on Networking (ToN)*, 15(5):1109–1122, 2007. 28, 29, 30

[LM97] TV Lakshman e Upamanyu Madhow. The performance of tcp/ip for networks with high bandwidth-delay products and random loss. *Networking, IEEE/ACM Transactions on*, 5(3):336–350, 1997. 2, 12

[LRW+17] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar et al. The QUIC transport protocol: Design and Internet-scale deployment. Em *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, páginas 183–196. ACM, 2017. 2, 31, 32

[LSM08] Douglas J. Leith, Robert N. Shorten e G. McCullagh. Experimental evaluation of cubic-tcp. Em *Proceedings of the 6th International Workshop on Protocols for Fast Long-Distance Networks (PFLDnet 2008)*, 2008. 26, 29, 30

[MGF+04] Saverio Mascolo, Luigi Alfredo Grieco, Roberto Ferorelli, Pietro Camarda e Giacomo Piscitelli. Performance evaluation of westwood+ tcp congestion control. *Performance Evaluation*, 55(1-2):93–111, 2004. 27, 29, 30

[mpt] Multipath tcp through a strange middlebox. http://blog.multipath-tcp.org/blog/html/2015/01/30/multipath_tcp_through_a_strange_middlebox.html. Acessado em: 24/03/2016. 20

[Nag84] John Nagle. Congestion Control in IP/TCP Internetworks. RFC 896, Janeiro 1984. 11

[Net] NetEm - Network Emulator. http://man7.org/linux/man-pages/man8/tc-netem.8.html. Acessado em: 17/09/2018. 30

[nin] The Ninja build system. https://ninja-build.org/manual.html. Acessado em: 17/09/2018. 34

[Not10] Mark Nottingham. HTTP Cache-Control Extensions for Stale Content. RFC 5861, Maio 2010. 11

[ns3] QUIC module for NS-3. https://gitlab.com/diegoamc/ns-3-quic-module. Acessado em: 17/09/2018. 33

[Pip] Http pipelining. https://en.wikipedia.org/wiki/HTTP_pipelining. Acessado em: 12/04/2016. ix, 15

[pla] Playing with QUIC. https://www.chromium.org/quic/playing-with-quic. Acessado em: 17/09/2018. 39

[Pos81] J. Postel. Transmission Control Protocol. RFC 793, Setembro 1981. 7, 9

[pro] Standalone QUIC library. https://gitlab.com/diegoamc/proto-quic. Acessado em: 17/09/2018. 33

[QUI] QUIC, a multiplexed stream transport over UDP. https://www.chromium.org/quic. Acessado em: 17/09/2018. 13

[RCC+11] Sivasankar Radhakrishnan, Yuchung Cheng, Jerry Chu, Arvind Jain e Barath Raghavan. Tcp fast open. Em *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, página 21. ACM, 2011. 8, 9

[RM15] Eric Rescorla e Nagendra Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347, Outubro 2015. 23

[Ros] Jim Roskind. QUIC: Design Document and Specification Rationale. https://docs.google.com/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34/edit. Acessado em: 21/07/2016. 2, 14

[RPB+12] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure e Mark Handley. How hard can it be? designing and implementing a deployable multipath tcp. Em *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, páginas 29–29. USENIX Association, 2012. 20

[Sha] Robbie Shade. Quic. next generation multiplexed transport over udp. http://bit.ly/1gqR7WN. Acessado em: 12/04/2016. ix, 16, 21

[Ste15] Randall R. Stewart. Stream Control Transmission Protocol. RFC 4960, Outubro 2015. 22

[Swe] Ian Swett. QUIC FEC v1. https://docs.google.com/document/d/1Hg1SaLEl6T4rEU9j-isovCo8VEjjnuCPTcLNJewj7Nk/edit. Acessado em: 21/07/2016. 20

[TC] tc - show / manipulate traffic control settings. https://linux.die.net/man/8/tc. Acessado em: 17/09/2018. 31

[TM02] Vassilis Tsaoussidis e Ibrahim Matta. Open issues on tcp for mobile computing. *Wireless Communications and Mobile Computing*, 2(1):3–20, 2002. 2, 12

[waf] The Waf build system. https://gitlab.com/ita1024/waf/. Acessado em: 17/09/2018. 34

[WHS15] Alyssa Wilk, Ryan Hamilton e Ian Swett. A QUIC Update on Google's Experimental Transport, 2015. https://blog.chromium.org/2015/04/a-quic-update-on-googles-experimental.html. Acessado em maio de 2018. 2

[WPY+04] Ren Wang, Giovanni Pau, Kenshin Yamada, MY Sanadidi e Mario Gerla. Tcp startup performance in large bandwidth networks. Em *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 2, páginas 796–805. IEEE, 2004. 2, 12

[XHR04] Lisong Xu, Khaled Harfoush e Injong Rhee. Binary increase congestion control (bic) for fast long-distance networks. Em *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 4, páginas 2514–2524. IEEE, 2004. 12

[XPMS01] George Xylomenos, George C Polyzos, Petri Mähönen e Mika Saaranen. Tcp performance issues over wireless links. *Communications Magazine, IEEE*, 39(4):52–58, 2001. 2

[YO02] John Yoakum e Lyndon Ong. An Introduction to the Stream Control Transmission Protocol (SCTP). RFC 3286, Maio 2002. 22