UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

RODRIGO DE SAPIENZA LUNA

MINITY: Um Framework para Testes de Protocolos de Rede

RIO DE JANEIRO
2021

RODRIGO DE SAPIENZA LUNA

MINITY: Um Framework para Testes de Protocolos de Rede

Trabalho de conclusão de curso de graduação apresentado ao Departamento de Ciência da Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Paulo Henrique de Aguiar Rodrigues, Ph.D.

RIO DE JANEIRO
2021

CIP - Catalogação na Publicação

L961m

Luna, Rodrigo de Sapienza
MINITY: um Framework para Testes de Protocolos de Rede / Rodrigo de Sapienza Luna. -- Rio de Janeiro, 2021.
101 f.

Orientador: Paulo Henrique de Aguiar Rodrigues.
Trabalho de conclusão de curso (graduação) - Universidade Federal do Rio de Janeiro, Instituto de Matemática, Bacharel em Ciência da Computação, 2021.

1. Controle de congestionamento. 2. Avaliação. 3. TCP. 4. Framework. 5. BBR. I. Rodrigues, Paulo Henrique de Aguiar, orient. II. Título.

Elaborado pelo Sistema de Geração Automática da UFRJ com os dados fornecidos pelo(a) autor(a), sob a responsabilidade de Miguel Romeu Amorim Neto - CRB-7/6283.

RODRIGO DE SAPIENZA LUNA

MINITY: Um Framework para Testes de Protocolos de Rede

Trabalho de conclusão de curso de graduação apresentado ao Departamento de Ciência da Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 19 de Novembro de 2021

BANCA EXAMINADORA:

Paulo Henrique de Aguiar Rodrigues
Ph.D (Instituto de Computação - UFRJ)

Silvana Rossetto
D.Sc (Instituto de Computação - UFRJ)

Valeria Menezes Bastos
D.Sc (Instituto de Computação - UFRJ)

Ao meu amor que me acompanhou em toda a minha trajetória, Letícia Ecard.

AGRADECIMENTOS

Ao meu amor Letícia, que esteve ao meu lado em todos os momentos desde o nosso ensino médio no Colégio Pedro II. Estar ao seu lado me fez amadurecer e sempre a ter forças para não desistir. Obrigado meu amor!

Aos meu pais, Carlos e Sheila, e aos meus irmãos Rafael e Gabriel, obrigado pelo apoio.

Aos meus sogros, Adilson e Rogéria, e a minha cunhada Karine, pelo suporte e carinho ao longo desses anos.

Ao meu orientador, Paulo Henrique de Aguiar Rodrigues pelo suporte e dedicação para a relização deste trabalho. Além de ser um excepcional professor e orientador, foi um grande amigo e conselheiro.

As professoras Silvana Rosseto e Valéria Bastos por participarem da banca de avaliação deste trabalho de conclusão de curso.

Aos professores do Instituto de Computação que tive o prazer de conhecer durante a graduação. Em especial a professora Márcia Cerioli, coordenadora do grupo de extensão para a maratona de programação, grupo pelo qual fiz parte durante toda a minha graduação e foi um divisor de águas para a minha formação.

Aos amigos que fizeram parte desses longos anos da graduação.

A todos que contribuiram para a minha formação acadêmica, muito obrigado.

“A ciência nunca resolve um problema sem criar pelo menos outros dez."

George Bernard Shaw

RESUMO

Desde o desenvolvimento do controle de congestionamento clássico do protocolo TCP, na década de 80, foram propostas diversas soluções para este problema. Recentemente, a Google propôs o TCP BBR, seguindo uma nova abordagem de controle de congestionamento baseado em taxa com a medição do Round-Trip Time e da banda disponível no gargalo da conexão. Para avaliar o comportamento de um protocolo é imprescindível experimentá-lo em diversos cenários a fim de garantir a sua corretude. Entretanto, a Google prototipou o BBR baseado em dados de tráfegos de sua rede interna, a qual dispõe de características diferentes da Internet pública. Devido a isso, uma nova versão do TCP BBR está em desenvolvimento pela Google a fim de mitigar possíveis vícios no projeto. Analisar o comportamento de um protocolo é uma tarefa árdua devido à dificuldade em simular o comportamento real de um protocolo complexo. Geralmente utilizam-se ferramentas de simulação, como o *ns-3*, que geram dados sintéticos de tráfegos de uma rede, mas que precisam já ter um modelo do protocolo já incorporado no ambiente. Apesar de se obterem excelentes estatísticas através dessas simulações o comportamento do tráfego real se difere de um modelo bem comportado. Devido a isso, ferramentas que são capazes de emular ambientes de redes e rodar códigos prototipados e operacionais em ambiente real são úteis para uma análise mais precisa. Neste trabalho de conclusão de curso é proposto e desenvolvido o framework MINITY com a finalidade de emular diversos cenários de rede para análise comparativa de diversas versões de TCP, com ampla flexibilidade de configuração de parâmetros como a distribuição de perda e variacão de latência e banda nos enlaces. O MINITY usa como base o ambiente MININET. Para a garantia do pleno funcionamento do MINITY e demonstração de suas pontencioalidades, os seus resultados foram comparados com os de outro framework, também baseado em MININET, mas sem as funcionalidades contempladas no MINITY.

Palavras-chave: tcp. controle de congestionamento. bbr. framework. simulação.

ABSTRACT

Since the development of the classical congestion control of the TCP protocol in the 80s, several solutions have been proposed for this problem. Google recently proposed TCP BBR, following a new congestion control approach based on Round-Trip Time and band measurement available at connection bottleneck. To evaluate the behavior of a protocol, it is essential to experiment with it in various scenarios in order to guarantee its correctness. However, Google prototyped BBR based on traffic data from its internal network, which has different characteristics from the public Internet. As a result, a new version of TCP BBR is being developed by Google in order to mitigate possible project biases. Analyze the behavior of a protocol is an arduous task due to the difficulty in simulating real environments. Simulation tools, such as *ns-3*, are generally used to generate synthetic traffic data from a network. In spite of the excellent statistics being obtained through these simulations, the behavior of real traffic differs from a well-behaved model. Therefore, tools that emulate network environments are quite useful for a more accurate analysis of a protocol's behavior. In this course completion work, the MINITY framework is proposed and developed to emulate several network scenarios for comparative analysis of different versions of TCP, under a quite plentiful of configuration link parameters as loss distribution and variation of latency and rate. MINITY is based on MININET. In order to ensure the full operation of MINITY, its results were compared to another framework, also based on MININET, but lacking the flexibility and versatility of MINITY.

Keywords: tcp. congestion control. bbr. framework. simulation.

LISTA DE ILUSTRAÇÕES

Figura 1 – Cabeçalho TCP. *Fonte: TANEBAUM 2011* 22
Figura 2 – Exemplo do funcionamento do algoritmo New Reno com múltiplas perdas e uma janela de congestionamento com 10 pacotes. Após entrar na fase de *rápida retransmissão* (Fast Retransmit), New Reno recupera os pacotes perdidos em 4 RTTs, linhas azuis. Além disso, mesmo durante a fase de recuperação, após o recebimento de ACKs duplicados a janela de congestionamento cresce e pacotes não enviados anteriormente são transmitidos. 26
Figura 3 – Evolução da janela de congestionamento dos algoritmos: BIC VS CUBIC. Fonte: (HA; RHEE; XU, 2008) 29
Figura 4 – Exemplo do Ponto máximo de operação ideal: *Kleinrock Point* Fonte: (CARDWELL et al., 2017) 31
Figura 5 – Exemplo de utilização da interface CLI disponibilizada pela API da MININET. Em a) pode-se observar através da CLI da MININET a abertura de um emulador de terminal *xterm* para que se possa acessar ao Sistema Operacional de uma das máquinas virtuais, com o nome "h1", criada pela MININET. Em b) o terminal "xterm"da máquina virtual com o nome "h1". 36
Figura 6 – Diagrama do *Handler* 38
Figura 7 – Diagrama do *Analyzer* 39
Figura 8 – Diagrama UML da classe Node. 40
Figura 9 – Exemplo de uso do utilitário tcpdump. São coletados os pacotes TCP da interface de entrada "enp2s0", informações de IP e porta da fonte e do destino podem ser observadas. 41
Figura 10 – Exemplo do utilitário tc. Neste exemplo informações do buffer de entrada da interface enp2s0 são coletadas. A quantidade de dados que estão enfileirados são informados através dos bytes no campo backlog. . 42
Figura 11 – Exemplo de uso do utilitário ss. Através do sufixo -tin"pode-se capturar os endereços IP e porta da origem e do destinatário. Além disso valores das variáveis do TCP também estão disponíveis. Neste exemplo observa-se que o sistema operacional estava operando com o TCP BBR e as variáveis de controle específicas como: mrtt, pacing gain e cwnd gain podem ser capturadas. 42
Figura 12 – Diagrama UML da Classe Edge. 43

Figura 13 – Exemplificação do escalonamento da interface de rede do Linux. O pacote é recepcionado pela interface de entrada e são policiados, assim, pacotes indesejáveis são descartados. Caso o pacote seja de destino interno, será entregue para as camadas superiores. Senão, o pacote é enviado para o roteamento ou é oriundo de camadas superiores, assim, é enfileirado na saída, por onde podem ser atrasados, marcados, descartados ou priorizados, e enviados para a interface de saída. Imagem extraída de: (Edgar Jamhour, 2015) 44
Figura 14 – A disciplina principal é chamada de *root* no comando tc. Além desta fila podem existir outras disciplinas pertencentes a classes de tráfego. A patir da distinção do tráfego de pacotes em uma interface, pode-se aplicar o controle de tráfego diferenciado para cada tipo de pacote que transita pela interface. Imagem extraída de: (Edgar Jamhour, 2015) . 45
Figura 15 – Hierarquia das qdics utilizadas pelo utilitário tc. A qdisc_1 *root* é a principal. As *qdisc* subsequentes são filhas das anteriores. Imagem extraída de: (University of South California, 2019) 45
Figura 16 – Exemplificação da árvore de hierarquia das qdiscs disponíveis para o controle de tráfego no sistema operacional Linux, através do utilitário tc. A qdisc com o handle 1: é a principal. Já as qdiscs subsequentes são suas filhas e filhas das classes. Observe a composição do handle das classes filhas, são compostos sempre pelo formato (handle do pai):(código da classe). Imagem adaptada de (HUBERT et al., 2002) 46
Figura 17 – Esquematização do algoritmo Token Bucket Filter para o controle de banda em uma disciplina de enfileiramento. O pacote recebido é enfileirado caso a fila não esteja cheia, caso ao contrário, é descartado. Os tokens são criados na velocidade em que se deseja transmitir o pacote, assim, quando um pacote só será transmitido caso a quantidade de tokens para sua transmissão esteja disponível. A fim de garantir que pacotes recebidos em rajadas não sejam descartados, o algoritmo define a quantidade de tokens que serão criados instanteamente. Imagem extraída de: (University of South California, 2019) 48
Figura 18 – Exemplo da configuração padrão de uma interface específica do framework MINITY. 48
Figura 19 – Diagrama UML da Classe Network. 49
Figura 20 – Exemplificação das interfaces de entrada e saída de uma transmissão. 50
Figura 21 – Formato Experimento JSON. Neste arquivo configura-se o experimento que será executado pelo framework MINITY. Através dele, todas as ações de alterações de banda, rtt e outras ações serão configurados. . . 53
Figura 22 – Exemplo de arquivo no formato json 54

Figura 23 – Exemplo de configuração de um Node. O Hospedeiro "h1"é um cliente e possui latência de 100ms. Ademais informações da fila de pacotes da interface de entrada serão coletados. 55
Figura 24 – Exemplo de como executar os métodos dos Gráficos do Módulo Coletor. Em (a) observa-se um código com a chamada da função run() para a execução de todos os gráficos do módulo Coletor; em (b) a sua execução a partir do Python 3 para a geração dos gráficos. 59
Figura 25 – Representação da topologia Dumbbell. Através dela, N hospedeiros servidor transmitem dados para N hospedeiros cliente através de três comutadores. 60
Figura 26 – Configuração dos Hospedeiros do experimento relatado. Em a) e b) representa a configuração, respectivamente, dos hospedeiros fontes h1 e h2. Já em c) e d) reporta a configuração dos hospedeiros destinos h3 e h4. 61
Figura 27 – Configuração dos três comutadores utilizados no experimento. 62
Figura 28 – Configuração dos Enlaces 64
Figura 29 – Arquivo de parâmetro em que se configura as ações do experimento . . 65
Figura 30 – Taxa de envio entre dois fluxos TCP BBR. Essa taxa é capturada a partir da informação dos pacotes TCP que passam pelo gargalo da rede. É realizada uma estimativa média de quantos bits/segundo estão sendo transmitidos a partir do tamanho dos pacotes TCP. 67
Figura 31 – Fila de pacotes do gargalo, na interface de saída do comutador 2. Duas filas estão presentes no gargalo da conexão, uma para o controle da taxa de envio de pacotes e outra para a adição de latência, taxa de perda e enfileiramento dos pacotes. Percebe-se que apenas 12000 bytes foram utilizados pelo protocolo TCP BBR. Isso se deve a descartes estarem ocorrendo na primeira fila, a de controle de banda. Mesmo assim, pode-se observar o TCP BBR utilizou apenas 12000 bytes, sempre constantes. Esse resultado se difere de outros protocolos, que dão uma sobrecarga maior na fila. Por fim, durante o acréscimo do RTT, houve uma redução da quantidade de pacotes enfileirados. 69
Figura 32 – Fila de pacotes do gargalo, na interface de saída do comutador 2. Duas filas estão presentes no gargalo da conexão, uma para o controle da taxa de envio de pacotes e outra para a adição de latência, taxa de perda e enfileiramento dos pacotes. Pode-se perceber que 15000 bytes foram utilizados pelo protocolo TCP NewReno, valor limite para a quantidade de rajadas aceitas inicialmente pelo controle de tráfego de MINITY. Ademais, após aumento do RTT, mais pacotes foram armazenados na segunda fila do gargalo da conexão. 70

Figura 33 – Fila de pacotes do gargalo, na interface de saída do comutador 2. Percebe-se que o TCP Cubic manteve a fila em torno de 15000 bytes. 70
Figura 34 – Variáveis de Controle do protocolo BBR. Em (a) encontra o gráfico da janela de congestionamento do TCP, dado pela quantidade de segmentos ao longo do experimento. Em (b) demonstra a variação da variável do Ganho da Janela de Congestionamento. Já em (c) pode-se observar o gráfico da banda estimada pelo BBR vs a banda utilizada pelos fluxos. 72
Figura 35 – Variáveis de Controle do protocolo BBR. Em (a) encontra-se a variável de medição do RTT vs o RTT real dos fluxos. Em (b) a variável pacing gain dos fluxos. 73
Figura 36 – Variáveis de Controle e Medições realizadas através do framework MINITY. Em (a) o resultado da janela de congestionamento. Em (b) o resultado da banda do gargalo da conexão vs valor de banda estimado pelo protocolo BBR. 76
Figura 37 – Variáveis de Controle e Medições realizadas através do framework MINITY. Em (a) encontra-se o resultado da medição do RTT (MRTT) vs RTT real da rede. Em (b) a taxa de envio calculado a partir da quantidade média de pacotes em um determinado intervalo de tempo. . 77
Figura 38 – Comparação entre os resultados da taxa de envio do framework MINITY com o framework desenvolvido em (JAEGER et al., 2019). . . . 79
Figura 39 – Comparação do resultado da alteração do RTT entre os frameworks MINITY e (JAEGER et al., 2019). 80
Figura 40 – Comparação dos resultados entre os frameworks. O objetivo é identificar o comportamento do protocolo BBR em um cenário com fluxos com diferentes RTTs. 82
Figura 41 – Comparação dos resultados entre os frameworks MINITY e o desenvolvido por (JAEGER et al., 2019) de um experimento com 6 fluxos TCP BBR divididos em dois grupos de RTT distintos com 40 ms e 80 ms. . 83
Figura 42 – Método run sendo executado com o parâmetro de abertura da interface do terminal para realizar a configuração manual da interface do gargalo. 91
Figura 43 – Comandos utilizados para a configuração do controle de tráfego nas interfaces do servidor e do comutador da conexão. 92
Figura 44 – Resultados do experimento ao utilizar o protocolo BBR. Em (a) observa-se a taxa de envio, referente ao valor médio da transferência da conexão em um intervalo de 0.08 segundos. Já em (b) a figura refere-se a quantidade de bytes na fila de saída do servidor. Por fim, em (c) o resultado da fila do gargalo da conexão, extraído do comutador. 95

Figura 45 – Resultados do experimento ao utilizar o protocolo NewRENO. Em (a) observa-se a taxa de envio, referente ao valor médio da transferência da conexão em um intervalo de 0.08 segundos. Já em (b) a figura refere-se a quantidade de bytes na fila de saída do servidor. Por fim, em (c) o resultado da fila do gargalo da conexão, extraído do comutador. 96
Figura 46 – Resultados do experimento ao utilizar o protocolo CUBIC. Em (a) observa-se a taxa de envio, referente ao valor médio da transferência da conexão em um intervalo de 0.08 segundos. Já em (b) a figura refere-se a quantidade de bytes na fila de saída do servidor. Por fim, em (c) o resultado da fila do gargalo da conexão, extraído do comutador. 97
Figura 47 – Resultado do experimento com 2 fluxos TCP BBR compartilhando um gargalo com 10 Mbit/s. O tamanho do gargalo foi configurado para suportar 500 pacotes. 99
Figura 48 – Resultado do experimento com 2 fluxos TCP CUBIC compartilhando um gargalo com 10 Mbit/s. O tamanho do gargalo foi configurado para suportar 500 pacotes. 100
Figura 49 – Resultado do experimento com 2 fluxos TCP RENO compartilhando um gargalo com 10 Mbit/s. O tamanho do gargalo foi configurado para suportar 500 pacotes. 101

LISTA DE ABREVIATURAS E SIGLAS

TCP	Transmission Control Protocol
UDP	User Datagram Protocol
BBR	Bottleneck Bandwidth and Round-trip propagation time
CWND	Congestion Window Size
IP	Internet Protocol address
RTT	Round Trip Time
MSS	Maximum Segment Size
SSTRESH	Maximum Segment Size
ACK	Acknowledgment
BDP	Bandwidth and Delay Product
API	Application Programming Interface
CLI	Command Line Interface
JSON	JavaScript Object Notation
BtlBW	Bottleneck Bandwidth
RTprop	Round Trip Propagation

SUMÁRIO

1 INTRODUÇÃO **16**
1.1 TEMA 16
1.2 Delimitação 16
1.3 Justificativa 17
1.4 Objetivo 18
1.5 Organização 19

2 TRANSMISSION CONTROL PROTOCOL - TCP **21**
2.1 Controle de congestionamento 21
2.2 Versões TCP 24
2.2.1 TCP NEW RENO **24**
2.2.2 TCP CUBIC **27**
2.2.3 TCP BBR **30**
2.2.3.1 *Startup* 31
2.2.3.2 *Drain* 31
2.2.3.3 *Probe Bandwidth* 32
2.2.3.4 *Probe* RTT 32
2.2.4 Evolução do TCP BBR **32**

3 FRAMEWORK MINITY **34**
3.1 Ambiente de Emulação de Rede 34
3.1.1 Flexibilidade **35**
3.1.2 Aplicabilidade **36**
3.1.3 Interatividade **36**
3.1.4 Escalabilidade **37**
3.1.5 Realismo **37**
3.1.6 Compartilhamento **37**
3.2 Estrutura do Framework 37
3.2.1 Características Gerais **38**
3.2.2 Bloco *Handler* **38**
3.2.2.1 Classe ***Node***: Configuração dos hospedeiros 39
3.2.2.2 Classe ***Edge***: Armazenamento de informações das conexões 42
3.2.2.3 Classe ***Network***: Configuração dos parâmetros de rede 48
3.2.2.4 Classe ***Minity*** 51
3.2.2.5 Classe Handler 52
3.2.2.5.1 Arquivos de Configurações **54**

3.2.2.6 Rotina de Configuração 55
3.2.3 Bloco *Analyzer* **56**
3.2.3.1 Módulo *Extrator* 57
3.2.3.2 Módulo *Coletor* 57
3.2.3.3 Módulo Plotagem 58

4 EXEMPLO DE USO DO MINITY **60**
4.1 Topologia 60
4.2 Configuração 60
4.3 Arquivo de Parâmetro 64
4.4 Análise de Resultados 66
4.4.1 Taxa de Envio e Fila de Pacotes **66**
4.4.2 Variáveis de Controle do Protocolo **71**

5 MINITY EM AÇÃO: PERFORMANCE COMPARATIVA TCP BBR **74**
5.1 Um único Fluxo BBR 74
5.1.1 Comparação entre os resultados **78**
5.2 Múltiplos Fluxos BBR 80
5.2.1 2 Fluxos BBR **81**
5.2.2 6 Fluxos BBR **83**

6 CONCLUSÃO E TRABALHOS FUTUROS **85**

REFERÊNCIAS **87**

APÊNDICE A – SIMULAÇÃO DE BUFFERBLOAT. **91**

APÊNDICE B – SIMULAÇÃO DE BUFFERBLOAT - GARGALO COM 6*BDP **98**

1 INTRODUÇÃO

1.1 TEMA

Desde o surgimento da internet, na década de 80, a sua utilização experimentou inúmeras transformações. As primeiras aplicações eram quase exclusivamente de âmbito acadêmico, e o conteúdo trafegado era, majoritariamente, de arquivos textos. Esse cenário modificou-se ao longo dos anos, e, atualmente, diversos são os recursos circulantes na rede, desde textos até arquivos de variadas extensões, como imagens, animações, vídeos, vídeos sob demanda e áudio. Além disso, características da própria rede, como velocidade, escalabilidade e segurança obtiveram melhorias ao longo dos anos. Devido a este fato, é inevitável que ocorram alterações nos protocolos da rede para que se ajustem a este comportamento.

O *Transmission Protocol Control* (TCP) é responsável pela transferência confiável de bytes entre os sistemas finais. Através do seu controle de congestionamento evita-se que um sistema final envie mais pacotes que a infraestrutura da rede possa suportar. Vários algoritmos de controle de congestionamento do TCP, com diferentes abordagens, foram propostos para atender aos requisitos das redes modernas. Entretanto, a avaliação e comparação de diferentes protocolos é uma tarefa árdua frente à diversidade dos cenários de rede por onde os pacotes de dados das aplicações transitam.

A fim de garantir corretude e desempenho, é necessário que os projetistas de protocolos avaliem suas propostas em variadas topologias de rede, submetidas a diferentes tráfegos e taxas de perda nos enlaces. Para a prototipação de algoritmos de controle de congestionamento do TCP algumas ferramentas de simulação são normalmente utilizadas, como a *ns-3* (RILEY; HENDERSON, 2010). Porém, ferramentas de simulação normalmente partem de modelos que incorporam uma simplificação do protocolo, podendo levar a resultados insuficientes para avaliar o comportamento real. Assim, ferramentas que sejam capazes de testar a implementação real de um protocolo em um cenário emulado de rede são imprescindíveis para garantir a confiabilidade e corretude das análises para protocolos em desenvolvimento.

1.2 DELIMITAÇÃO

Todas as camadas da arquitetura TCP/IP contêm protocolos, responsáveis por controlar e permitir que uma comunicação possa ocorrer entre dois sistemas computacionais. A camada de transporte, especificamente, é responsável pela transferência de dados entre duas máquinas, independentemente da aplicação e da configuração das redes físicas utilizadas para transmitir os pacotes. Dois protocolos principais pertencem a esta camada: o

UDP (POSTEL, 1980) e o TCP (POSTEL, 1981). Este trabalho de conclusão de curso conduzirá experimentos relacionados à comparação de diferentes versões de TCP com uso de diferentes controle de congestionamento para regulagem do tráfego ofertado.

1.3 JUSTIFICATIVA

Diversos algoritmos de controle de congestionamento para o TCP foram propostos ao longo dos anos. O TCP clássico, desenvolvido em (JACOBSON, 1988), tem o seu controle de congestionamento baseado no princípio de que é preciso transmitir com taxas crescentes até que haja uma detecção de congestionamento na rede, ocasionando em uma redução da taxa de envio pela metade. Após esta redução, o tráfego passa a ser novamente crescente, originando um comportamento conhecido como *dente de serra*. Este comportamento, que é padrão na Internet, tem como consequência natural a formação de longas filas de espera no gargalo e, em decorrência, há o aumento da latência das conexões que passam pelo gargalo.

Sabe-se que o ponto ótimo de operação do TCP, ou seja, a taxa ótima de envio que é função do tamanho da janela de transmissão, deveria ser na vazão limite permitida pelo gargalo. Devido ao controle de congestionamento do TCP clássico permitir que o ponto ótimo seja ultrapassado, como consequência do aumento do tráfego, há um crescimento na fila do gargalo até que ocorra uma perda. Fica evidente que o TCP clássico opera além do ponto ótimo, como abordado em (CARDWELL et al., 2017) e apresentado na Figura 4. Sendo assim, para que a taxa de transmissão do TCP opere ao redor do ponto ótimo é necessário um controle baseado em taxa e não em perda. O ponto ótimo é muito dinâmico por conta da variação de tráfego em uma rede e do número de conexões que se passam pelo gargalo ao longo do tempo. Como já provado por (JAFFE, 1981), criar um algoritmo distribuído para encontrar o ponto ótimo para a operação é impossível. Apesar desta impossibilidade teórica, várias propostas de controle baseadas em taxa têm sido encontradas na literatura.

Recentemente, a Google desenvolveu o TCP BBR (CARDWELL et al., 2017) seguindo uma nova abordagem, na qual o controle de congestionamento deve reagir a apenas ao congestionamento real da rede, desconsiderando seus indicadores, como aumento da latência e perda de pacotes. Para a prototipação do protocolo, porém, a Google utilizou informações de sua rede interna, o que pode causar vícios no projeto - devido às características da rede interna da Google não corresponderem com a realidade da rede global de computadores. Pode-se destacar, ainda, que a empresa está interessada em otimizar o atraso nas transferências de arquivos entre os seus servidores, caracterizando um cenário em que há milhares de conexões TCP abertas em paralelo entre duas máquinas, todas com RTT semelhante, operando com o mesmo TCP e compartilhando os mesmos gargalos. Esse cenário se difere de um gargalo na Internet, no qual se encontram milhares de

conexões com variados RTTs e com diferentes implementações de TCP, gerando preocupação com equidade e desempenho entre os vários usuários. A Internet pública apresenta um enorme desafio e os parâmetros escolhidos pela análise de sensibilidade da rede da Google podem não ser adequados ao caso geral de uso do TCP BBR combinado com o TCP clássico em conexões variadas, com diferentes RTTs e vazões. Como TCP BBR é um investimento de alguns anos da Google e a mais sólida e promissora implementação para TCP no momento, tendo a sua versão 1 disponibilizada, avançando para uma versão 2 do protocolo e em estudo pelo IETF de uma padronização (CARDWELL et al., 2019), urge um ambiente de testes que viabilize a busca de aprimoramentos e a escolha adequada dos parâmetros do protocolo para os cenários de uso geral.

A criação de cenários é uma tarefa árdua, na qual, inúmeras vezes, não se é possível averiguar o comportamento de um protocolo em cenários que poderiam ser problemáticos. Geralmente, esses protocolos são submetidos a testes através de simulações, com dados sintéticos, ou em um cenário bem específico com uma quantidade limitada de recursos.

Em (JAEGER et al., 2019), é proposto um framework capaz de simular um ambiente de rede para analisar o comportamento do TCP BBR em diversos cenários frente a outros algoritmos de controle de congestionamento do TCP. Esse framework dispõe de várias limitações como: a carência de modularização do código, tornando difícil a adição de novas funcionalidades; a ausência de uma interface para o usuário para configuração de um experimento; ausência de flexibilização da topologia de rede - apenas a topologia de dumbbell está disponível, em que N hospedeiros servidores transmitem dados para N hospedeiros clientes através de três roteadores ou comutadores, como visto na Figura 25; limitação na definição de parâmetros de rede - apenas alterações no Round-Trip Time e a taxa de banda do enlace são possíveis; e transferência de dados sem a utilização de um protocolo da camada de aplicação.

1.4 OBJETIVO

O objetivo deste trabalho foi propor e desenvolver MINITY, um framework de emulação de rede, também como o objetivo de analisar diferentes versões de TCP, mas dotado de maior flexibilidade de configuração, facilidade de operação e modular para facilitar extensões. Para isso, alguns requisitos foram considerados no desenvolvimento, como:

- modularizar o código para permitir que novas funcionalidades possam ser acopladas.
- criar uma interface para o usuário com a finalidade de permitir a configuração de um experimento sem a necessidade de alterar o código fonte.
- flexibilizar a construção da topologia para possibilitar personalização.
- prover transferência de dados reais pela rede através de um protocolo da camada de aplicação.

- agrupar a criação da topologia de rede e a análise de dados decorrente em um único ambiente.

O desenvolvimento de MINITY envolveu a integração do emulador de redes MININET (BRANDON HELLER, 2013) com diversas ferramentas disponibilizadas no Linux utilizadas para o gerenciamento da interface de rede. Além disso, para a criação de uma interface que permitisse o usuário construir o seu experimento, sem que seja necessário alteração no código fonte, utilizou-se o formato JSON. A maior parte do código foi escrita na linguagem Python e para a modularização do código utilizou-se a programação orientada a objetos. Com a emulação da rede sendo executada pela MININET, espera-se que seja possível emular uma rede com centenas e até milhares de hospedeiros e comutadores. Para a validação de MINITY utilizou-se cenários com até 6 fluxos com TCP BBR com diferentes RTTs e comparou-se os resultados obtidos em (JAEGER et al., 2019).

Por MINITY ser *open-source*, futuros desenvolvedores poderão aprimorá-lo, seja na adição de novas análises em protocolos que pertencem as camadas de aplicação, transporte ou rede, ou no aprimoramento das atuais análises. Ademais, devido à flexibilidade que MINITY propõe para a criação de cenários, ambientes de redes mais diversos podem ser modelados, como uma rede sem fio - através da adição de variação de latência e perda de pacotes durante a transmissão. Outro fator importante é a capacidade de MINITY na captação dos comportamentos específicos de um protocolo como TCP BBR, como a ocorrência de ciclos e a baixa pressão na fila do gargalo da conexão. Devido a suas características funcionais, MINITY pode ser considerado mais amigável e potencialmente mais útil que o framework desenvolvido por (JAEGER et al., 2019) e a diversas outras ferramentas de simulação, como a *ns-3* (RILEY; HENDERSON, 2010), principalmente por MINITY prover um ambiente completo com interface que facilita a usabilidade e análise de dados acoplada.

Para a melhor compreensão do trabalho realizado, espera-se que o leitor tenha um conhecimento mínimo de redes, envolvendo o funcionamento básico do protocolo TCP, como mostrado em livros textos tradicionais da área, como (KUROSE, 2013; TANENBAUM, 2011; COMER, 2016) para que se possa compreender o comportamento dos protocolos e as análises realizadas neste trabalho. Já para a compreensão da arquitetura e do desenvolvimento do framework MINITY, espera-se um conhecimento básico em programação com a linguagem Python e conceitos de programação orientada a objetos.

1.5 ORGANIZAÇÃO

Este documento é organizado em seis Capítulos e um Apêndice.

O Capítulo 1 apresenta a introdução, com a motivação e síntese dos principais resultados alcançados pelo framework.

No Capítulo 2 é feita uma revisão do TCP, com o foco nos principais TCPs como: Reno, New Reno, CUBIC e BBR.

O Capítulo 3 apresenta a arquitetura do framework MINITY, as classes que o compõe e os seus módulos.

No Capítulo 4 é descrito um exemplo de uso do MINITY, a fim de auxiliar os futuros usuários na preparação de um experimento e facilitar a primeira experiência com o framework.

No Capítulo 5 é realizada a avaliação do framework com a finalidade de validá-lo frente ao framework inspirador deste trabalho, através de três experimentos: o primeiro com um único fluxo BBR sofrendo alterações de banda e de RTT; o segundo com dois fluxos BBR com RTTs diferentes compartilhando um gargalo; e o terceiro com seis fluxos BBRs divididos em dois grupos com RTTs diferentes compartilhando um gargalo.

No Capítulo 6 se encontram as conclusões e sugestões de trabalhos futuros.

No Apêndice A utiliza-se o framework para a comparação entre os protocolos TCP BBR, TCP CUBIC e TCP NewRENO, frente ao errôneo dimensionamento da fila do gargalo da conexão.

2 TRANSMISSION CONTROL PROTOCOL - TCP

O *Transmission Control Protocol* foi definido formalmente na RFC 793 em 1981 (POSTEL, 1981) e tem sido aprimorado ao longo dos anos. Inúmeras versões com várias características distintas foram desenvolvidas e implementadas. O TCP é um protocolo da camada de transporte da arquitetura TCP/IP que tem o objetivo de realizar o transporte confiável de dados, mesmo que a camada subjacente de rede possua uma infraestrutura heterogênea, com topologias diversas, largura de banda, atrasos, tamanho de pacotes, diversos equipamentos, e outros parâmetros completamente diferentes, e seja eventualmente não confiável.

Uma conexão TCP entre dois hospedeiros pode ser vista como um duto confiável para a transferência bidirecional de dados, havendo a garantia da entrega íntegra e em ordem de todos os bytes da aplicação transferidos em ambos sentidos da conexão. Olhando apenas um sentido da transferência, o hospedeiro que envia os dados será o hospedeiro servidor, enquanto que o outro será o hospedeiro cliente. Os dados provenientes da camada de aplicação são segmentados em blocos e antecedidos de um cabeçalho para compor um segmento TCP, como apresentado na Figura 1. Este cabeçalho de formato fixo de 20 bytes pode ser seguido pelo campo de opções que possui um tamanho variável de no máximo 40 bytes. Os valores de portas origem e destino são usadas para identificar as aplicações que estão se comunicando, portanto, uma conexão TCP é identificada pela quádrupla: ip origem, porta origem, ip destino e porta destino. Os campos *Número de sequência* e *Número de confirmação* são utilizados para o controle de fluxo de dados, sendo respectivamente a identificação do segmento e para a indicação do próximo segmento esperado pelo destino. O campo *Ponteiro para urgente* é irrelevante por ter finalidade e não será detalhado. Já o campo *Tamanho da janela* é a variável rcwnd e indica o tamanho disponível do buffer do receptor que é utilizado para indicar ao transmissor uma redução no fluxo de transmissão de dados. Por fim o campo *Checksum* é utilizado para a detecção de falhas em segmentos específicos, logo, aumenta a confiabilidade durante a transmissão de dados.

Este capítulo está dividido em 2 seções. A primeira faz uma breve revisão do controle de congestionamento do TCP. A segunda descreve as principais versões do TCP: Reno, New Reno, Bic, Cubic e, por fim, o TCP BBR, que é usado para demonstrar o potencial da ferramenta MINITY desenvolvida.

2.1 CONTROLE DE CONGESTIONAMENTO

O congestionamento em uma rede de computadores ocorre quando a transferência de dados por esta rede é superior a sua capacidade de transmissão. Sendo assim, a lotação dos

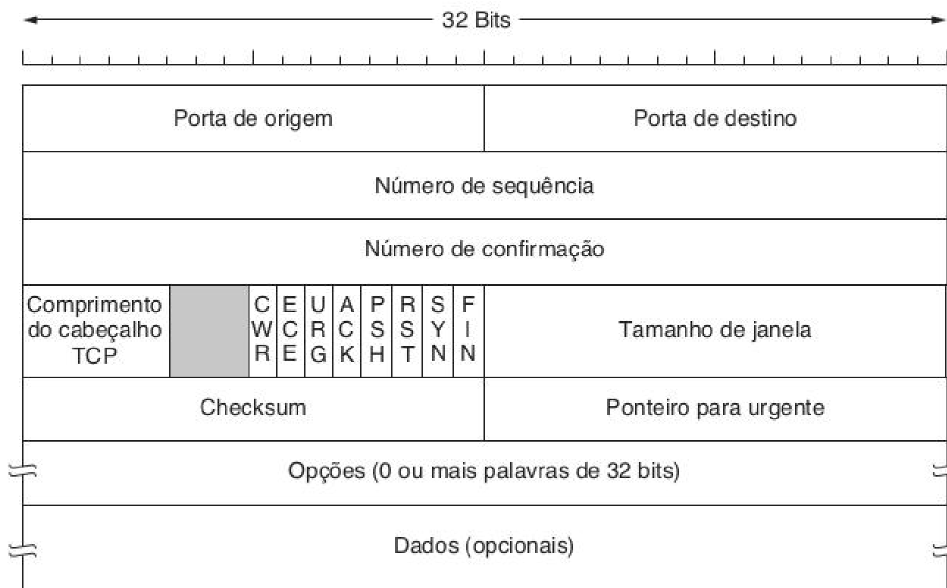


Figura 1 – Cabeçalho TCP.
Fonte: TANEBAUM 2011

buffers dos roteadores se apresenta como um dos fatores causadores de congestionamento, o que provoca um descarte de pacotes antes mesmo de serem processados e direcionados.

A primeira versão do TCP não possuía controle de congestionamento. Durante a década de 80, o problema do congestionamento foi identificado em (NAGLE, 1984) e ficou conhecido como **colapso por congestionamento**. Em (JACOBSON, 1988) foram propostos diversos algoritmos que solucionavam este problema através do controle de congestionamento e logo foram padronizados para o protocolo TCP (ALLMAN; PAXSON; BLANTON, 2009), esses algoritmos são: *slow start (SS)* ou partida lenta, *congestion avoidance (CA)* ou prevenção de congestionamento, *fast retransmit* ou retransmissão rápida, e *fast recovery* ou recuperação rápida. No TCP, esses algoritmos são utilizados em diversas fases durante a transmissão e são necessários para prover o controle de congestionamento fim a fim deste protocolo.

O mecanismo de controle de congestionamento, exercido pelo TCP, é operado pelo remetente através do monitoriamento de uma variável adicional, a janela de congestionamento. Esta variável, também denominada cwnd, é responsável por limitar a taxa de transmissão do remetente. Assim, após o remetente enviar a quantidade limite que a janela de congestionamento o permite, só poderá transmitir mais dados quando receber a confirmação de que os dados já enviados foram recebidos pelo destinatário. O número de bytes a serem transmitidos a qualquer instante não pode ser superior à janela rcwnd anunciada pela ponta receptora.

Em slow start, no início da conexão, opera-se com $cwnd$ = 1 MSS (pode-se usar até 4 MSS inicialmente em situações específicas), onde 1 MSS é o tamanho máximo de um segmento TCP. Nesta situação, o remetente só poderá enviar 1 segmento e ficará

aguardando o recebimento do ACK de confirmação deste primeiro segmento enviado, quando poderá incrementar a cwnd. A taxa de envio inicial é então 1 MSS/RTT. Como a rede pode possuir uma quantidade de banda disponível maior do que esta taxa inicial, a cada ACK recebido pelo remetente reconhecendo um segmento já transmitido, cwnd é incrementada de 1 MSS. Assim, nesta primeira fase, o aumento da janela é exponencial, pois a taxa de envio dobra a cada RTT (round trip time, o tempo de ida e volta entre as pontas da conexão TCP). O TCP permanece nesta fase até que uma perda seja detectada; ou exceder o limite de tempo (timeout) para que seja recebido a confirmação de um dos segmentos já enviados; ou até que a janela de congestionamento atinja um valor limite pré-definido e conhecido como sstresh; ou até que o remetente identifique uma perda através do recebimento de 3 ACKs duplicados.

Ao entrar no estado de prevenção de congestionamento, o valor de cwnd será sempre maior ou igual a sstresh. Nesta fase, o TCP opera de maneira conservadora e aumenta o tamanho da janela de congestionamento em 1 MSS a cada RTT, tentando de forma menos agressiva alcançar taxas crescentes para a conexão.

A reação do TCP difere em relação aos eventos que possam ser causados por perda de um segmento ou forte congestionamento na rede. Quando houver o estouro do limite de tempo de um segmento, o timeout, o TCP irá interpretar como uma indicação de forte congestionamento e o valor de sstresh é ajustado para a metade do valor atual da janela de congestionamento (cwnd) e o valor da janela de congestionamento é alterado para 1. Após a atualização dessas variáveis, o TCP reinicia no processo de partida lenta novamente e permanece até que cwnd iguale sstresh. Caso isso ocorra, o TCP entra em fase de prevenção de congestionamento (CA).

Caso haja o recebimento recebimento de 3 ACKs duplicados, o TCP assume que houve uma perda eventual de um segmento, sem ocorrência de forte congestionamento na rede. O TCP sai da fase de partida lenta ou de prevenção de congestionamento e entra na fase de rápida retransmissão (fast retransmit). Nesta fase, o TCP pode retransmitir o segmento perdido de forma imediata, reinicializando o tempo de expiração do segmento. Além disso, a janela de congestionamento é reduzida pela metade, e soma-se a ela 3 MSS, para incluir os 3 ACKs duplicados recebidos, e o valor da janela limite (sstresh) é reduzido para a metade do valor da janela de congestionamento, quando os 3 ACKs foram recebidos. Então o TCP entra na fase de recuperação rápida (fast recovery).

Na fase de recuperação rápida, a janela de congestionamento é incrementada em 1 MSS para cada ACK duplicado recebido para o segmento perdido que fez o TCP entrar nesta fase. Assim que um ACK confirmar o segmento sendo recuperado, o TCP atualiza o valor da janela de congestionamento para o valor da janela limite e entra na fase de prevenção de congestionamento. Entretanto, caso ocorra o timeout do segmento, o comportamento é igual ao o que ocorre na fase de partida lenta.

Logo, desconsiderando a fase inicial do TCP, a partida lenta, o comportamento do

controle de congestionamento do TCP consiste de aumentos lineares na janela de congestionamento por 1 MSS por RTT e decréscimos multiplicativos na janela de congestionamento (AIMD, *Additive-Increase Multiplicative-decrease*), formando um comportamento semelhante aos dentes de um serrote.

2.2 VERSÕES TCP

Diversos algoritmos de controle de congestionamento foram propostos para o TCP com o intuito de alcançar os seguintes objetivos: garantir a alta utilização da rede, evitar a formação de grandes filas, evitar a perda de pacotes e, por fim, garantir que a distribuição de banda entre os fluxos seja justa.

Os algoritmos como TCP NewReno (ALLMAN; PAXSON; BLANTON, 2009) e o CUBIC (HA; RHEE; XU, 2008) utilizam a possível perda de pacotes como indicativo de um congestionamento na rede. Deste modo, quando ocorre um evento de três ACKs duplicados, que poderia ser causado por uma perda eventual de um segmento ou chegada fora de ordem no destino, o TCP tradicional assume a perda como o evento mais provável e uma indicação de leve congestionamento na rede. Veja que a corrupção de pacotes nos enlaces ou nós é considerado evento de baixíssima probabilidade dada a confiabilidade dos equipamentos e dos meios de comunicação. O TCP interpreta então que está operando em uma taxa de transmissão maior do que o gargalo da rede suporta e que deva fazer uma diminuição da taxa de envio, diminuindo cwnd. Os protocolos clássicos divergem apenas na forma com que aumentam ou diminuem cwnd.

Nesta seção serão discutidas algumas versões dos algoritmos de controle de congestionamento do TCP para que seja possível realizar a comparação com o algoritmo que este trabalho deseja evidenciar, o BBR (CARDWELL et al., 2017).

2.2.1 TCP NEW RENO

O TCP RENO (ALLMAN; PAXSON; BLANTON, 2009) foi a primeira implementação clássica do controle de congestionamento do TCP proposto por (JACOBSON, 1988), ao utilizar as fases de *slow start*, *congestion avoidance*, *fast retransmit* e o *fast recovery*, como descrita na seção anterior. Apesar destes mecanismos melhorarem o desempenho do controle de congestionamento do TCP, o RENO não é eficiente no cenário em que ocorra múltiplas perdas (STOICA, 2005). Isso se deve ao seu comportamento em reduzir a janela de congestionamento a cada detecção de uma perda, logo, na decorrência de múltiplas perdas, a janela reduzirá pela metade múltiplas vezes. Além disso, o TCP RENO entra na fase de prevenção ao congestionamento, com aumentos lineares da janela de congestionamento, o que pode acarretar em uma subutilização da rede (WANG et al., 2004), após a redução severa da janela de congestionamento. Uma outra característica é o favorecimento de fluxos com RTTs pequenos (LAKSHMAN; MADHOW, 1997).

O TCP NewReno (HENDERSON et al., 2012) resolve o decaimento exponencial com o número de múltiplas perdas na janela cwnd do TCP RENO. Quando uma detecção de perda ocorrer após três ACKs duplicados, NewReno entra em fase de retransmissão rápida e após esta fase entra na fase de recuperação, assim como o TCP RENO. Entretanto, os protocolos se diferem na fase de recuperação. Enquanto o TCP RENO sai da fase de recuperação e entra na fase de prevenção ao congestionamento após o ACK do pacote retransmitido ser recebido, o TCP NewReno somente muda de fase quando receber um ACK que confirme todos os pacotes enviados até o momento da fase de retransmissão rápida, ou na ocorrência de um timeout. Para isso, utiliza o mecanismo de ACK parcial (HENDERSON et al., 2012) que confirma um número parcial de pacotes do transmissor quando entrou na fase de recuperação.

O recebimento de um ACK parcial causa a retransmissão do próximo pacote esperado pelo receptor da transmissão. Após o recebimento do ACK parcial, a janela de congestionamento é incrementada em 1 e reduzida do valor igual a soma do tamanho dos pacotes que acabaram de serem confirmados pelo ACK parcial. Além do mais, cada ACK parcial confirma o pacote que foi retransmitido e indica a próxima perda ocorrida na janela de transmissão, logo cada perda é recuperada em 1 RTT. Desta forma o TCP NewReno evita o problema da redução exponencial da janela de congestionamento, pois só entra na fase de prevenção ao congestionamento quando todas os pacotes perdidos até a fase de retransmissão sejam confirmadas pelo receptor.

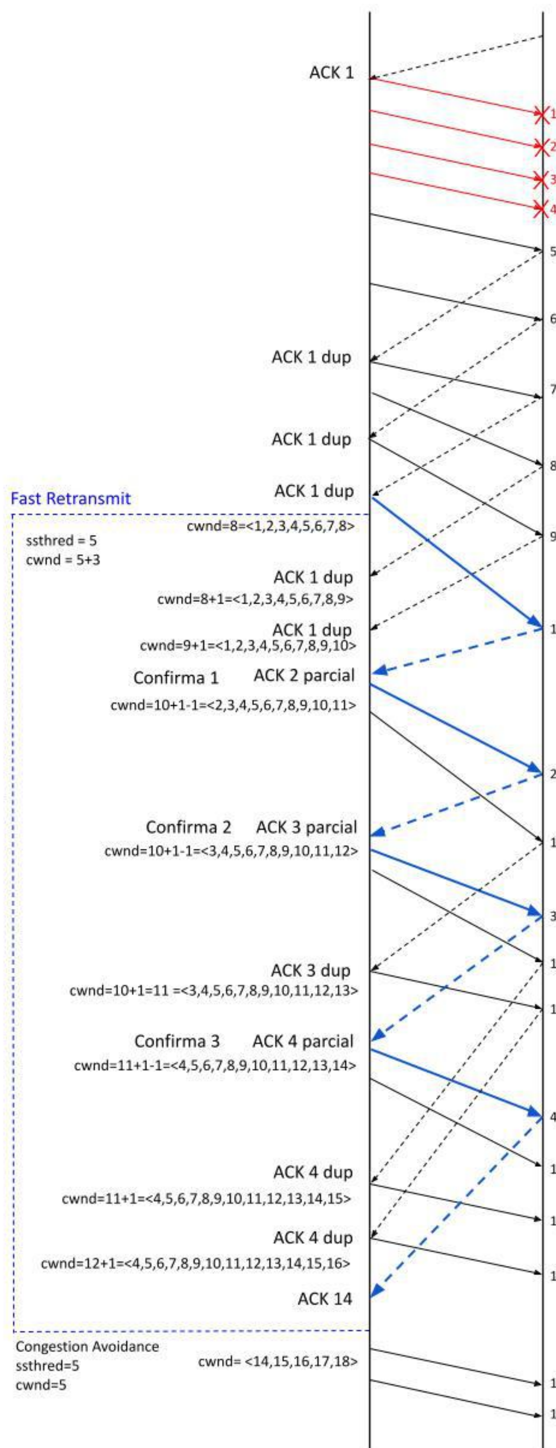


Figura 2 – Exemplo do funcionamento do algoritmo New Reno com múltiplas perdas e uma janela de congestionamento com 10 pacotes. Após entrar na fase de *rápida retransmissão* (Fast Retransmit), New Reno recupera os pacotes perdidos em 4 RTTs, linhas azuis. Além disso, mesmo durante a fase de recuperação, após o recebimento de ACKs duplicados a janela de congestionamento cresce e pacotes não enviados anteriormente são transmitidos.

A Figura 2 descreve uma transmissão de segmentos através do TCP NewReno com

a finalidade de exemplificar o seu funcionamento em um cenário com múltiplas perdas. Nesta transmissão a janela de congestionamento possui tamanho 10 e os segmentos enumerados de 1 a 10 são transmitidos, no entanto, os segmentos de 1 até 4 são perdidos. A partir do momento em que se recebe os 3 ACKs duplicados para o segmento 1, pode-se observar, que o TCP entra na fase de rápida retransmissão, ajusta o limite da janela de congestionamento (sstresh) para 5 e reduz a janela de congestionamento (cwnd) para 8, metade da janela anterior adicionado 3 unidades, para os ACKs duplicados do segmento 1 e reenvia este segmento antes que ocorra o timeout. Antes que receba o ACK parcial confirmando a retransmissão do segmento 1, 2 ACKs duplicados são recebidos para este mesmo segmento e consequentemente cada ACK duplicado incrementa a janela de congestionamento em 1 unidade. Observa-se que a partir do momento em que novos segmentos surgem durante o aumento da janela de congestionamento, eles são enviados imediatamente. Após 1 RTT o remetente recebe o ACK parcial confirmando o recebimento da retransmissão do segmento 1, além de indicar que o segmento 2 também precisa ser retransmitido. Nesse momento, o segmento 2 é retransmitido e a janela de congetionamento é incrementada em 1 unidade e decrementado pelo total de segmentos confirmados pelo ACK parcial, ou seja, 1 unidade. A janela de congestionamento é atualizada, e o segmento 1 é removido da janela e o segmento 11 é adicionado a janela, e devido a este segmento não ter sido enviado anteriormente, ele é transmitido. Após o recebimento do ACK parcial confirmando que o recebimento do segmento 2, ele é removido da janela de congestionamento e o segmento 12 é adicionado e transmitido para o destinatário.

Podemos perceber que esse comportamento ocorre, até que os 4 ACKs parciais, para os segmentos perdidos no início da transmissão, sejam recebidos. Após a confirmação de todos os segmentos enviados antes da fase de retransmissão rápida, o TCP NewReno entra na fase de prevenção ao congestionamento, com janela de congestionamento de tamanho 5.

Percebe-se que o TCP NewReno logrou êxito ao evitar uma redução exponencial, proporcional ao número de segmentos perdidos, na janela de congestionamento. Além disso, a fase de recuperação durou cerca de 4 RTTs para retransmitir os segmentos perdidos e manteve o envio novos segmentos de acordo com os ACKs duplicados recebidos pelo remetente. Por fim, após a fase de recuperação, a quantidade de segmentos a serem enviados nesta fase de prevenção de congestionamento são mínimos, evitando-se rajadas de pacotes.

2.2.2 TCP CUBIC

BDP (*Bandwidth and Delay Product*) é o produto do atraso ida e volta vezes a taxa da conexão. Este produto indica a quantidade de bits que poderiam trafegar durante o intervalo de tempo de RTT segundos. Para obter a máxima utilização de uma conexão TCP é necessário que o tamanho da janela de congestionamento (cwnd) seja da ordem do

BDP da conexão. Se for assim, no instante que todos os segmentos permitidos pela janela de congestionamento foram transmitidos, ocorre a chegada de um ACK que confirma a base da janela - o segmento da janela com o menor número de sequência e ainda não confirmado. Este ACK confirma a base e permite o deslize da janela e o envio de um novo segmento, mantendo a máxima utilização possível. Se o cwnd for menor que o BDP da conexão, então haverá a interrupção da transmissão até a chegada da confirmação da base, o que deixa a conexão ociosa e é claramente uma ineficiência.

Essa ineficiência está presente no controle de congestionamento clássico do TCP, visto que a fase de prevenção ao congestionamento incrementa a janela de congestionamento de 1 MSS a cada RTT segundos (*Round Trip Time*). Isso se torna um problema em redes de alta velocidade, pois poderá deixar a rede em baixa utilização por um longo período de tempo até que a janela de congestionamento se adeque ao BDP da rede. Seja, por exemplo, uma rede com largura de banda de 10 Gbps e um RTT de 120ms, valor médio entre São Paulo/BR e New York/USA (WONDER NETWORK, 2021). Considerando que os segmentos possuem o tamanho de 1250 bytes, então o BDP desta rede é em torno de 210 mil segmentos. Ao operar na fase de prevenção ao congestionamento, o crescimento a partir da metade do BDP (100 mil segmentos) leva cerca de 100 mil RTTs para que se chegue ao ponto de máxima utilização da rede, algo que demora cerca de 12000 segundos ($\approx 3,3 horas$). Este tempo é muito longo e há a necessidade de alterar o comportamento do TCP para permitir que a janela de congestionamento cresça mais rapidamente e, ao mesmo tempo, permitir que o tamanho da janela em caso de perda seja alterado para um valor acima da metade da janela. A combinação destas duas ações faz com que a inclinação do dente de serra seja mais agressiva, permitindo uma taxa média mais elevada para a conexão TCP e período menor para o dente de serra.

Uma primeira proposta de sucesso para alterar o comportamento do TCP clássico para redes com BDP elevado foi o BIC. O algoritmo TCP-BIC (XU; HARFOUSH; RHEE, 2004) solucionava o problema de lentidão da janela de congestionamento ao alcançar uma saturação máxima da rede através de uma busca binária. No momento em que se é detectado uma perda, o valor atual de cwnd é salvo como o MaxCWND e, então, utiliza o último tamanho de cwnd, que obteve a perda, e o define como MinCWND. Como esses dois pontos representam uma função monotonicamente crescente é possível utilizar a busca binária para encontrar um ponto ótimo de saturação. A janela de operação utilizada será o ponto médio entre MinCWND e o MaxCWND. Caso alguma perda ocorra, o valor de MaxCWND é então ajustado para o valor atual de cwnd, e o ponto médio entre MinCWND e MaxCWND ajustado será utilizado como o novo ponto de operação, esse procedimento, ocorre até que o remetente encontre um ponto ótimo, para a janela de congestionamento, estável. O comportamento desse algoritmo é ótimo para a recuperação da máxima utilização da rede e entrega uma alta estabilidade para a transmissão.

Em uma rede, inúmeros fluxos de transmissão coexistem e qualquer protocolo de trans-

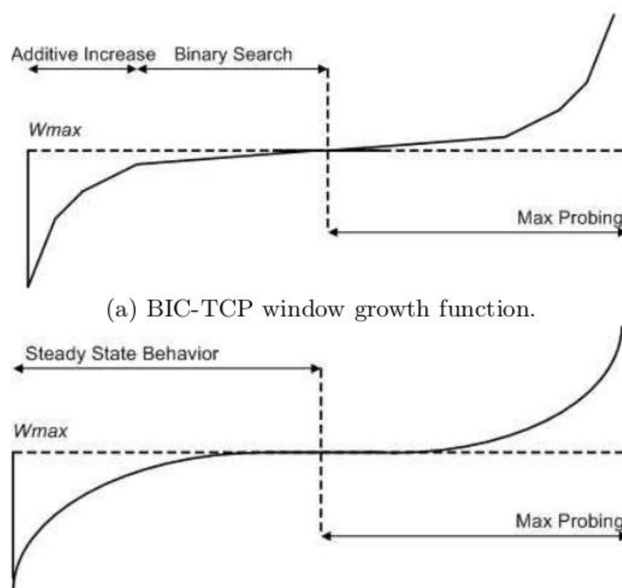


Figura 3 – Evolução da janela de congestionamento dos algoritmos: BIC VS CUBIC.
Fonte: (HA; RHEE; XU, 2008)

porte deve estar atento a distribuição igualitária da banda disponível, independentemente do algoritmo utilizado ou das condições de rede. Devido a rápida convergência, do BIC, para um ponto de saturação ótimo, um fluxo com baixo RTT pode ser beneficiado e em contrapartida um fluxo com alto RTT é prejudicado na distribuição da banda disponível. Além disso, fluxos com outros algoritmos de controle de congestionamento são prejudicados devido a esta rápida convergência. O que faz com que o TCP-BIC seja desigual ao compartilhar os recursos da rede, entre fluxos com RTT e protocolos diferentes.

TCP CUBIC (HA; RHEE; XU, 2008) é uma melhoria do algoritmo BIC (XU; HARFOUSH; RHEE, 2004), tendo sido adotado como padrão no sistema operacional LINUX em 2006. O controle de congestionamento do CUBIC é uma função cúbica bastante similar ao modo do funcionamento do BIC. Na Figura 3 podemos observar o comportamento similar entre ambos algoritmos, com a diferença na suavidade da curva e na tentativa de alcançar um novo ponto máximo de saturação, *Max Probing*, e dispõe de uma curva mais controlada ao invés de uma forma linear em relação ao comportamento do TCP BIC. CUBIC utiliza uma função cúbica em função do tempo decorrido desde o último evento de congestionamento. A função da janela de congestionamento do CUBIC utiliza a Equação 2.1, onde C é um parâmetro cúbico, t é último instante em que ocorreu uma redução na janela de congestionamento e K, como encontrado na Equação 2.2, é o tempo que a função demora para ir do valor W para Wmax. A constante β na Equação 2.2 é o fator de decremento da janela de congestionamento quando ocorre uma perda, geralmente, utiliza-se o valor de β em 0.25, logo, após uma perda, a nova janela de congestionamento

seria de $\frac{3}{4}$ do valor anterior.

$$W(t) = C(t - K)^3 + W_{\max} \tag{2.1}$$

$$K = \sqrt[3]{\frac{W_{\max} * \beta}{C}} \tag{2.2}$$

O comportamento deste algoritmo é possuir uma função de crescimento côncava e convexa, e em alguns momentos durante a sua execução, utiliza as características desse tipo de função. No início de um fluxo, o CUBIC têm uma rápida convergência para algum valor utilizando a parte côncava da função e após atingir esse objetivo altera o comportamento para a parte convexa.

CUBIC, por ser uma melhoria do BIC, resolve as suas respectivas ineficiências e tem um melhor compartilhamento dos recursos da rede. Devido à independência da sua taxa de crescimento de janela de congestionamento em relação ao RTT do fluxo, CUBIC não prioriza fluxos com diferentes RTTs e é justo com outros fluxos TCP.

2.2.3 TCP BBR

O controle de congestionamento clássico do TCP implementa um algoritmo que reage a perda com a diminuição da janela de congestionamento. Portanto, o controle de congestionamento clássico precisa da ocorrência do evento de perda, detectado através de 3 ACKs duplicados ou por timeout.

No entanto, esse controle de congestionamento clássico dispõem de dois problemas principais. Primeiro, é a suscetibilidade à perda aleatória de pacotes, uma vez que o TCP interpreta a perda como um congestionamento e reduz a taxa de transmissão por algum fator, levando a subutilização da rede. Em segundo, operam acima do ponto de máxima utilização da rede, *Kleinrock point*, e são sensíveis ao tamanho dos buffers do gargalo. Na Figura 4 podemos observar que o ponto de operação desses algoritmos dependem do tamanho do buffer do gargalo e quando esses buffers são grandes, ocasiona o problema de *BufferBloat* (GETTYS, 2011), o que provoca o aumento da latência em todas as conexões que utilizam este gargalo.

TCP BBR (CARDWELL et al., 2017) propõe uma nova abordagem de controle de congestionamento baseada no congestionamento. Ao invés do algoritmo utilizar indicadores de congestionamento para o controle da taxa de transmissão, o TCP BBR realiza medições da largura de banda do gargalo (BtlBW) e do *round-trip propagation time* (RTprop) para identificar se um caminho está congestionado, a fim de evitar que filas sejam formadas nos roteadores. Com isso, o BBR garante que o gargalo da conexão sempre se mantenha em torno do ponto de máxima saturação, *Kleinrock point*, mas sem que causar congestionamento.

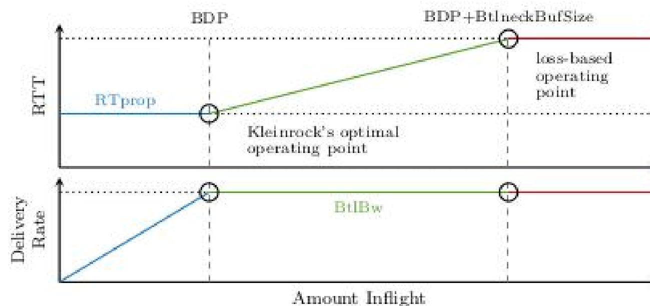


Figura 4 – Exemplo do Ponto máximo de operação ideal: *Kleinrock Point*
Fonte: (CARDWELL et al., 2017)

BBR utiliza o histórico recente de RTTs e da quantidade de dados confirmados pelo destinatário para estimar o tempo de propagação e a largura de banda do gargalo. No entanto, esses valores não podem ser medidos simultaneamente, pois a tentativa de aumentar taxa de transmissão de dados faz com que filas se formem no gargalo, o que aumenta o RTT. Devido a isso, essas variáveis são medidas separadamente.

Para controlar a quantidade de dados enviados, o BBR utiliza uma variável que indica o fator que se deve incrementar ou decrementar do BtlBW para o controle da taxa de envio, chamado *pacing gain*.

O Algoritmo BBR têm 4 fases diferentes: *Startup*, *Drain*, *Probe Bandwdth* e *Probe* RTT.

2.2.3.1 *Startup*

No ínicio da conexão o BBR não tem informações para poder estimar as variáveis BtlBW e RTprop. Por este motivo, a cada ACK recebido será dobrado a quantidade de envio de pacotes, assim como ocorre no TCP CUBIC. A partir do momento em que se possa realizar a medição das variáveis, esse incremento exponencial não é utilizado e o BBR assume que alcançou o ponto máximo de saturação.

2.2.3.2 *Drain*

Nesta fase, devido ao incremento exponencial na fase *Startup*, o BBR assume que filas foram criadas. Então, para drená-las, o algoritmo, temporariamente, reduz o *pacing gain* para o inverso da taxa de aumento utilizada na fase *Startup*. Quando o total de pacotes que foram enviados e não foram confirmados é igual a taxa estimada do BDP, significa que as filas foram completamente drenadas, mas o gargalo ainda está com a sua máxima utilização. Quando esse evento ocorre, o BBR passa da fase *Drain* para próxima, a *Probe Bandwidth*.

2.2.3.3 *Probe Bandwidth*

O TCP BBR opera na maior parte do tempo nesta fase, com o objetivo de aumentar o fluxo de de pacotes durante a transmissão. Ela é composto por oito ciclos, com a duração de um RTprop cada. No ínicio do ciclo, o algoritmo tenta aumentar a quantidade de dados transmitos com o ajuste do *pacing gain* para 1.25, seguido por uma drenagem de fila, com o ajuste do *pacing gain* para 0.75. Isso deve a possibilidade da criação de filas durante o processo de aumentar a taxa de envio, para isso, é necessário reverter qualquer fila criada com o ajuste proporcinal ao aumento de banda. No restante dos seis ciclos o BBR ajusta o *pacing gain* para 1. A amostragem da banda é realizada constantemente e é utilizada como estimador de banda disponível no gargalo (BtlBW). A duração desta fase é de dez segundos, para que seja realizado a medição do RTprop.

2.2.3.4 *Probe* RTT

Como a medição do RTprop não pode ser realizada, simultaneamente, com a medição da banda, após 10 segundos o BBR entra na fase ProbeRTT. Nesta fase, o algoritmo reduz a banda para quatro pacotes, para que seja drenada qualquer possível fila criada pela fase anterior. O objetivo é obter a estimação do RTT mais real possível. Esta fase é mantida por 200ms mais um RTT. Após a medição, se um valor mínimo é encontrado o RTprop então é atualizado para este valor. O RTprop medido terá validade de dez segundos e, assim, o BBR entra na fase de *Probe Bandwidth*. Esse ciclo é mantido até que a conexão seja encerrada.

2.2.4 Evolução do TCP BBR

A primeira versão do TCP BBR foi prototipado pela Google com o objetivo de otimizar a transferência de dados entre os seus servidores de sua rede interna. Esse cenário se difere com o cenário da Internet Pública por apresentar milhares de conexões TCPs longas, paralelas e com o mesmo valor de RTT. Assim parâmetros como: o tempo de duração na fase *Probe RTT*; a quantidade de ciclos em uma mesma fase; valores das variáveis *pacing gain* e *congestion gain* podem não ser adequados para o cenário da Internet Pública. Por isso, a Google está desenvolvendo a versão 2 do BBR (CARDWELL et al., 2019) a fim de mitigar possíveis vícios do projeto.

A versão 2 do projeto tem como foco aperfeiçoar a coexistência do TCP BBR com outros protocolos TCP como o New Reno e o CUBIC, reduzir a pressão nas filas a fim de reduzir o descarte de pacote e o atraso, melhorar a convergência da estimação do RTT e reduzir a variância da vazão de banda. Definir os parâmetros do protocolo para garantir essas melhorias é uma tarefa complexa, pois diversos cenários devem ser considerados. Como por exemplo, definir o tempo para a realização de uma nova medição do RTT - na versão 1 demora 10 segundos, valor bem alto para fluxos curtos. Ademais estudos

demonstraram que a versão 1 do TCP BBR sofre com a sensibilidade ao RTT, como pode ser visto em (JAEGER et al., 2019; MA et al., 2017).

Definir adequadamente os valores dos parâmetros requer um exaustivo trabalho na aferição dos parâmetros do TCP BBR em diversos cenários. Assim, ambientes que proporcionam a criação de cenários para que seja possível averiguar o comportamento de protocolos se tornam bastante promissores.

3 FRAMEWORK MINITY

Com o desenvolvimento do algoritmo BBR (CARDWELL et al., 2017), proposto pelo Google, tornou-se oportuno desenvolver um sistema que fosse capaz de criar um ambiente de emulação e extrair métricas das transmissões de pacotes com a finalidade de analisar o comportamento de protocolos. O Google está desenvolvendo a versão 2 do protocolo TCP BBR (CARDWELL et al., 2019) e visa empregá-lo em sua rede interna para efetuar testes, experimentações e prototipagens. Entretanto, empregar os testes em um domínio específico de aplicação, como o Youtube, pode acarretar na prototipação de um protocolo com um viés pela ausência de testes em outros cenários reais. Portanto, é visível a necessidade de um framework que seja capaz de simular diferentes cenários de redes de maneira realística e que tenha uma interface amigável para o usuário.

Em (JAEGER et al., 2019) foi proposto um framework para explorar cenários de aferição de desempenho do BBR. No entanto, a ferramenta proposta apresenta várias limitações: disponibiliza apenas uma única topologia de rede, o código fonte não é modularizado, não utiliza um protocolo de transferência de arquivos entre os hospedeiros, não é possível adicionar novas funcionalidades de uma maneira simples, e possui limitação na configuração dos *hosts*, *switches* e dos parâmetros de rede. Este framework foi a inspiração e base inicial para o desenvolvimento do framework MINITY.

O desenvolvimento de MINITY teve como objetivo desenvolver uma ferramenta versátil e com uma interface simples e amigável. Para facilitar a incorporação de novas funcionalidades à ferramenta e ainda manter uma programação simples e modular, foi escolhida a orientação a objetos para a implementação. E em relação à facilidade de utilização da ferramenta por terceiros, houve a preocupação de construir uma interface que recebesse a parametrização da rede, sem a necessidade da escrita de linhas de código e/ou programação em liguagem de scripts ou semelhante. Com isso, o uso da ferramenta por outros não apresenta qualquer dificuldade e permite a reprodutibilidade de testes e experimentos para validação de resultados alcançados por novas propostas e/ou conjecturas de melhorias aos protocolos. Além disso, há ampla facilidade para modificações e adições ao ambiente da ferramenta por um desenvolvedor qualificado. Essas características são uma melhoria significativa em relação ao framework descrito em (JAEGER et al., 2019).

No presente capítulo, abordaremos a arquitetura do framework MINITY, seu desenvolvimento, as ferramentas utilizadas em sua construção e a estrutura do código.

3.1 AMBIENTE DE EMULAÇÃO DE REDE

Para emular um ambiente de rede no framework MINITY, foi utilizado o sistema emulador de redes *open-source* MININET (KAUR; SINGH; GHUMMAN, 2014a), que

possibilita a simulação de uma rede completa, composta por *hosts*, *switches*, roteadores, controladores e links. Essa ferramenta providencia os recursos necessários para emular redes complexas em uma única CPU. Através de comandos simples, é possível realizar alterações e adicionar novas configurações na rede virtual. MININET emprega uma forma mais leve de virtualização a nível de processo, já que muitos recursos do sistema são compartilhados. Desta maneira, o emulador permite uma alta escalabilidade do sistema, sendo possível simular milhares de *hosts* e *switches*, alterando apenas uma única variável.

A emulação permite utilizar uma implementação real de um protocolo para a geração de tráfego em uma rede simulada, fazendo com que os dados obtidos sejam muito próximos do que seria em uma rede física real. Além disso, no ambiente de rede simulado, há a possibilidade de alteração de parâmetros como taxas de perdas nos enlaces e envolver outros tráfegos de interesse. A emulacão de rede se torna essencial para estressar protocolos frente a diferentes cenários de rede. E, mais importante, a utilização da implementação real do protocolo em máquinas virtuais permite a transferência imediata para um uso operacional, com plena validação da implementação.

O projeto MININET disponibiliza uma API em python (BRANDON HELLER, 2013) que proporciona ao desenvolvedor uma plataforma simples, com ampla flexibilidade para a criação de um experimento sobre topologias complexas. MININET possui uma interface que recebe comandos de linha (CLI) para realizar testes e debugs das topologias e é capaz de compartilhar os experimentos, códigos e testes com outros desenvolvedores. Esta ferramenta é comumente utilizada em pesquisas, desenvolvimento de aplicações, criação de topologias extensas, entre outras tarefas que possuam a necessidade da emulação de um ambiente de rede real. Devido às qualidades citadas e outras, encontradas em (KAUR; SINGH; GHUMMAN, 2014b), a ferramenta MININET tem sido utilizada em dezenas de instituições, incluindo Princeton, Berkeley, NASA, Stanford, startups, em algumas universidades no Brasil (LANTZ; HELLER; MCKEOWN, 2010), e também utilizada no trabalho que inspirou o framework MINITY (JAEGER et al., 2019).

O emulador MININET foi desenvolvido a partir dos conceitos de flexibilidade, aplicabilidade, interatividade, escalabilidade, realismo e compartilhamento. Através de seu sistema de virtualização, permite a criação, a interação, a customização e compartilhamento de topologias criadas. Portanto ao incorporar essa ferramenta no framework MINITY, esses benefícios são integrados.

Explicitaremos, a seguir, de uma forma mais ampla e detalhada, os benefícios proporcionados pela utilização do emulador MININET.

3.1.1 Flexibilidade

A flexibilidade oferecida pela MININET permite que novas topologias e características sejam configuradas neste emulador ao utilizar a API disponível do sistema (BRANDON HELLER, 2013). Por exemplo, ao criar uma topologia com o uso da MININET é possível

modificar as configurações, como configurações de um enlace e adicionar/remover conexões e nós da rede.

Outra característica inerente à MININET é a sua capacidade de se adaptar e utilizar os recursos do sistema operacional vigente. Um exemplo é a possibilidade de empregar o controle de tráfego do *kernel* do Linux, chamado TC (HUNBERT, 2001), para controlar o tráfego entre os hospedeiros, enlaces e roteadores. Assim, pode-se alterar configurações de mecanismos de qualidade de serviço (QoS) do sistema operacional do hospedeiro ou roteador em tempo de execução do experimento.

3.1.2 Aplicabilidade

A MININET entrega versatilidade na simulação do protótipo de rede através de ferramentas e interfaces de simples manuseio pelo projetista de rede. Logo, antes de ocorrer a implantação física que requer decisão de investimentos na compra de hardware e compromisso de atender os critérios de desempenho e qualidade de experiência para os usuários, a topologia projetada pode ser testada, estressada e validada através da MININET, permitindo o ajuste de configurações e especificações, otimizando custos e evitando desperdício financeiro. Uma topologia de rede que tenha sido previamente avaliada e testada garante que os objetivos desejados serão atingidos plenamente.

3.1.3 Interatividade

Enquanto a aplicabilidade é a facilitadora pela construção do sistema, a interatividade proporcionada pela MININET é responsável por um contínuo ambiente de gerenciamento para o usuário. Assim, durante um experimento em execução, o usuário pode interagir com a ferramenta, efetuar modificações ou averiguar se o experimento está sendo conduzido da maneira que foi planejado.

A API da MININET (BRANDON HELLER, 2013) disponibiliza, em python, uma interface de linha de comando (CLI), que permite ao usuário a possibilidade de configurar, gerenciar e testar o projeto da rede. Por exemplo, pode-se utilizar o terminal, xterm do Linux, para abrir uma interface do host desejado, bastando utilizar o comando *xterm hostname* como visto na figura 5. Assim o projetista pode averiguar a sua funcionalidade, suas configurações e até alterar uma configuração, através desta interface.

(a) Interface CLI

(b) Terminal xterm

Figura 5 – Exemplo de utilização da interface CLI disponibilizada pela API da MININET. Em a) pode-se observar através da CLI da MININET a abertura de um emulador de terminal *xterm* para que se possa acessar ao Sistema Operacional de uma das máquinas virtuais, com o nome "h1", criada pela MININET. Em b) o terminal "xterm"da máquina virtual com o nome "h1".

3.1.4 Escalabilidade

Na concepção de uma ferramenta de emulação de redes é importante o fato dela ser escalável, ou seja, ser possível emular uma topologia com uma quantidade variável de nós e recursos de rede. Devido a virtualização dos hospedeiros, *switches* e roteadores da rede emulada ocorrer em uma única CPU, existe a limitação dos recursos disponíveis. Logo, para experimentos simulados que utilizam uma rede de larga escala, seu desempenho dependerá proporcionalmente do hardware utilizado pelo sistema que esteja emulando essa rede virtual. Entretando, devido a emulação ocorrer através de um processo de virtualização leve em que os recursos são compartilhados, através da MININET é possível emular uma rede de grande escala, com dezenas e até milhares de nós.

3.1.5 Realismo

O ambiente emulado deve representar o mundo real com um alto grau de confiabilidade. Então, as aplicações e pilhas de protocolos em uso no mundo real devem ser utilizadas sem a adição de qualquer modificação. Através dos recursos disponíveis nos sistemas operacionais e da capacidade da MININET de emular via software os hospedeiros, equipamentos e as suas funcionalidades, é possível, então, simular com alto grau de confiabilidade um ambiente real através da MININET.

3.1.6 Compartilhamento

Em algumas ocasiões é interessante o compartilhamento do protótipo com outros colaboradores para que executem, modifiquem e realizem testes em qualquer ambiente. A MININET foi desenvolvida para ser uma ferramenta com capacidade de ser facilmente compartilhada com outros usuários.

3.2 ESTRUTURA DO FRAMEWORK

O framework MINITY tem como propósito ser um ambiente controlado para a realização de análises em redes e em protocolos de comunicação. O desenvolvimento do framework MINITY foi estruturado para permitir que novas funcionalidades possam ser adicionadas sem grandes alterações no código fonte. Diferentemente do framework desenvolvido em (JAEGER et al., 2019), no qual foi desenvolvido um framework para analisar apenas o protocolo TCP BBR em uma única topologia, no framework MINITY optou-se por um ambiente flexível com acoplamento de diversos testes e experimentos em um único ambiente de testes.

Para isso, MINITY foi projetado a partir do modelo de orientação a objetos e as suas funcionalidades foram organizadas em classes. Dessa maneira, um desenvolvedor pode acoplar uma funcionalidade à plataforma com facilidade. Além disso, o framework foi

segmentado em dois blocos: um para a construção do experimento e outro para a análise, como será descrito adiante.

Nesta seção será abordada a implementação e as funcionalidades oferecidas pela estrutura do framework MINITY.

3.2.1 Características Gerais

O framework MINITY possui dois blocos com objetivos claros e distintos: o *Handler* e o *Analyzer*. O bloco *Handler* é responsável pela construção da topologia e do experimento a ser executado pelo o agente *gerenciador* a partir de dois arquivos, *topologia.json* e *experimento.json*, como visto na Figura 6. A coleta de dados é realizada a partir de scripts python que são executados pelas máquinas virtuais, sendo realizada uma amostragem das *features* necessárias. O bloco *Analyzer*, representado pela Figura 7, por sua vez, tem como objetivo principal estruturar os dados coletados e extrair informações do experimento, explorando métricas de desempenho pré-definidas, através de três módulos: *Extrator*, *Coletor* e *Plotagem*. O módulo *Extrator* define quais são os arquivos, parâmetros e características necessárias para que o módulo *Coletor* realize a tarefa de extrair as informações a partir dos dados brutos. Por fim, o módulo *Plotagem* contém a geração dos gráficos das métricas pré-definidas.

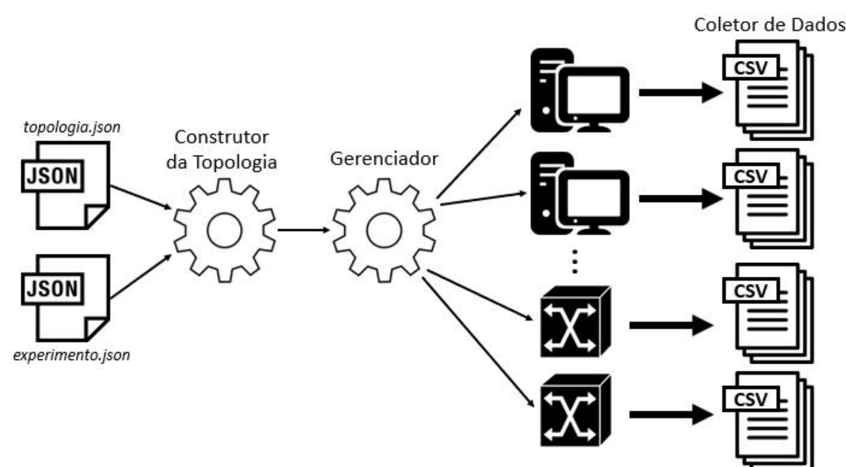


Figura 6 – Diagrama do *Handler*

3.2.2 Bloco *Handler*

Nesta seção será descrito a formalização da implementação das classes utilizadas no desenvolvimento do framework MINITY, seus objetivos e características. As classes descritas por esta seção são relacionadas ao bloco *Handler*, que tem como objetivo configurar o ambiente e executar o experimento.

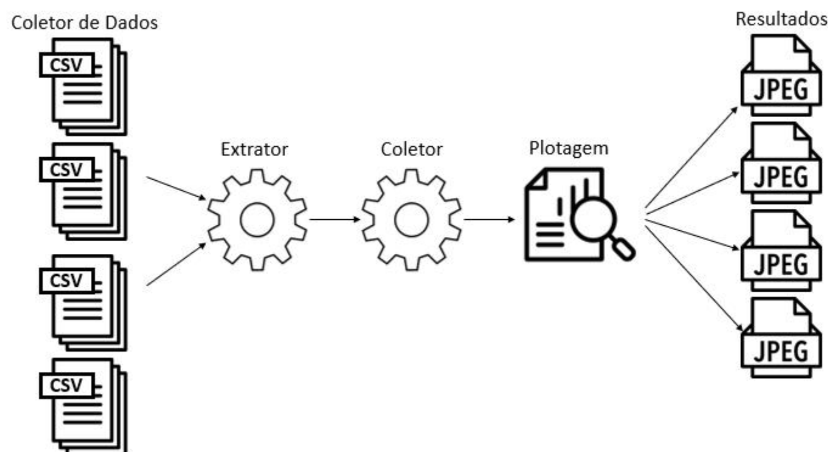


Figura 7 – Diagrama do *Analyzer*

A fim de prover suporte à construção do framework MINITY, foi utilizada a API da MININET (BRANDON HELLER, 2013). E, devido à necessidade da criação de novas funcionalidades, foram implementadas novas classes herdeiras da API original.

Para a concepção da topologia, foi utilizado a modelagem a partir do conceito de grafo, o qual é formado por dois conjuntos V e E. Segundo (SZWARCFITER, 1983) um grafo $G = (V, E)$ é formado por um conjunto finito não-vazio V e um conjunto E de pares não-ordenados de elementos distintos de V, dentre quais, os elementos de V são chamados de vértices e os elementos pertencentes a E são chamados de arestas. Sendo assim, na abstração utilizada, os vértices serão elementos tais como: hospedeiros, *switches* e roteadores. Em contra-partida, os links são definidos como as arestas nesta modelagem.

As classes ***Node*** e ***Edge*** são a representação dos vértices e arestas do grafo, respectivamente. Cada objeto que pertence à classe ***Node*** possui informações de parâmetros, como configurações de fila (*queue*) do Linux; parâmetros de outras classes das quais ele herda objetos, como a classe ***Sniffer***; e configurações de Servidor de FTP.

A classe ***Edge*** agrega informações relacionadas à conexão entre os dois *hosts*, como a informação da largura de banda, o tamanho do buffer, as interfaces de comunicação entre os dois hosts, taxa de perda de pacote, a variação de atraso (*jitter*) e até mesmo a função distribuição da perda de pacotes.

3.2.2.1 Classe *Node*: Configuração dos hospedeiros

A classe ***Node***, representada na Figura 8, contém as informações relacionadas aos hospedeiros e suas variáveis de configuração. Cada hospedeiro deve possuir algumas características específicas, como, por exemplo, a capacidade de transferir arquivos entre si através da rede disponibilizada pela MININET e capturar os dados destas transferências. Desse modo, avaliou-se a necessidade da existência de classes para implementar essas funcionalidades, com o objetivo de permitir a flexibilidade na adição de novos componentes. Portanto, a fim de manter a flexibilidade do framework, a classe ***Node*** é modelada como

uma classe que herda atributos e métodos das classes ***Sniffer*** e ***FTP***, responsáveis pela amostragem de dados e pelo protocolo de transferência de arquivos, respectivamente.

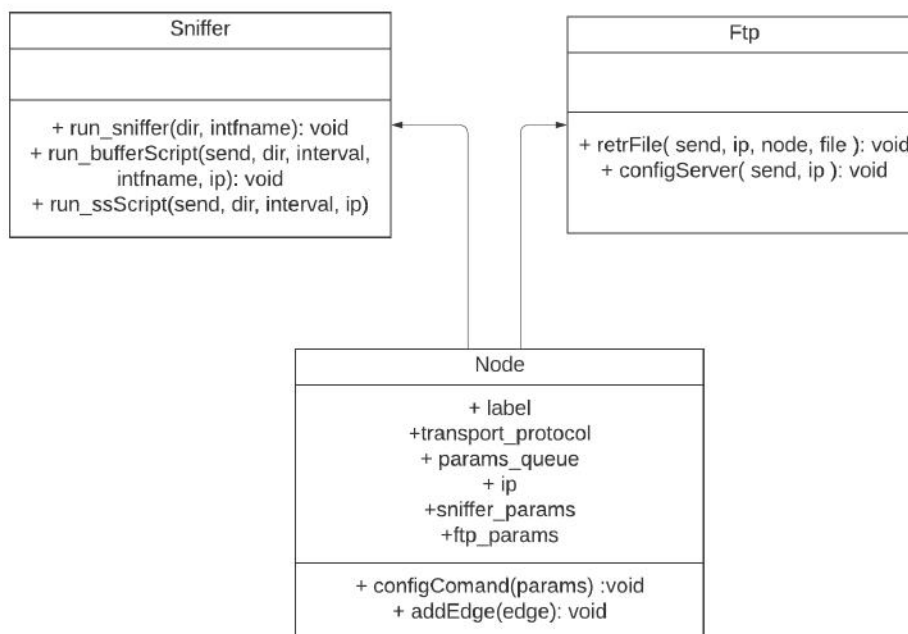


Figura 8 – Diagrama UML da classe Node.

Essa classe possui duas variáveis, denominadas label e IP, que o usuário precisa definir com atenção. A variável label, que é o identificador do hospedeiro, é utilizada para a referenciar o objeto que contém todas as informações de um hospedeiro. A variável IP, que representa o endereço IP do hospedeiro, deve ser definida de acordo com a topologia de rede de seu experimento e respeitando as restrições de criação de sub-redes IP. Devido ao framework MINITY ter como objetivo a experimentação de protocolos TCP e não a avaliação de topologias gerais de Internet, o desenvolvimento da ferramenta assumiu que todos os hospedeiros estão conectados em uma rede local comutada e fazem parte de uma mesma sub-rede IP com máscara 255.0.0.0. ou /8.

O modelo de referência TCP/IP, ao contrário do modelo OSI/ISO de sete camadas, adota uma estrutura simplificada envolvendo apenas as camadas de aplicação, transporte, rede, enlace e física. A camada de aplicação engloba protocolos que definem a comunicação entre as aplicações e, por outro lado, a camada de transporte tem o papel de prover o serviço para a comunicação entre os processos de aplicação residentes em computadores distintos. Para realizar a comunicação pela rede disponibilizada pela MININET foi

utilizado o protocolo de aplicação FTP (POSTEL; REYNOLDS, 1985) que utiliza TCP para a transferência confiável de arquivos entre hospedeiros.

Para incorporar o FTP no framework MINITY, adicionamos uma classe que implementa este protocolo a partir de duas bibliotecas *open source* disponibilizadas para a linguagem Python: pyftpdlib (GIAMPAOLO RODOLA, 2008) e ftplib (GUIDO VAN ROSSUM, 1992). A primeira é usada para configurar o servidor FTP no hospedeiro servidor e a segunda permite que o hospedeiro cliente gere requisições ao hospedeiro servidor FTP através de conexões TCP providas pelo protocolo de transporte alvo.

Percebe-se que para a adição de qualquer outro protocolo de aplicação, com o objetivo de analisá-lo ou analisar o comportamento dos protocolos subsequentes, é imprescindível a criação de uma classe que irá implementá-lo, mantendo a característica de modularização do framework MINITY, que utiliza a programação orientada a objetos. Por exemplo, para analisar o HTTP/3 é necessário um servidor WEB, como o APACHE, e uma classe HTTP deverá ser implementada com o objetivo de configurar os servidores e as suas respectivas requisições. Além disso, é necessário implementar alterações no arquivo JSON de parâmetros e de configuração do framework.

A classe ***Sniffer***, por sua vez, teve a sua concepção idealizada com o propósito de agregar funções que realizem a extração de dados durante os experimentos, sejam dados da comunicação entre os hospedeiros ou dados de variáveis internas ao hospedeiro. O monitoramento da comunicação entre os hospedeiros é efetuado a partir de bibliotecas que podem ser instaladas em qualquer ambiente UNIX. Para a extração de métricas relevantes à observação do comportamento das versões do TCP foram empregadas três ferramentas: *tcpdump* (Van Jacobson, Craig Leres and Steven McCanne, 2010), *tc* (ALEXEY N. KUZNETSOV, BERT HUBERT , 2001) e *ss* (ALEXEY N. KUZNETSOV, 2001). O programa tcpdump, comumente utilizado para monitorar pacotes trafegados em uma interface de rede, é aplicado pelo framework MINITY para extração de dados referentes aos pacotes TCP, para análise posterior. A Figura 9 representa os comandos que definem os parâmetros para a escolha da interface e filtragem dos pacotes. Assim, ao utilizar os comandos *-i* e *tcp*, apenas pacotes TCP serão extraídos pela ferramenta na interface especificada.

```
(base) root@rodrigoluna-Inspiron-7472:~# tcpdump tcp -i enp2s0 -n
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on enp2s0, link-type EN10MB (Ethernet), capture size 262144 bytes
19:03:00.688128 IP 192.168.1.4.60632 > 104.76.108.117.443: Flags [P.], seq 2294659360:2294659399, ac
k 2957897091, win 501, options [nop,nop,TS val 577574598 ecr 1043451035], length 39
19:03:00.828894 IP 104.76.108.117.443 > 192.168.1.4.60632: Flags [P.], seq 1:40, ack 39, win 640, op
tions [nop,nop,TS val 1043457069 ecr 577574598], length 39
19:03:00.828954 IP 192.168.1.4.60632 > 104.76.108.117.443: Flags [.], ack 40, win 501, options [nop,
nop,TS val 577574739 ecr 1043457069], length 0
19:03:01.132394 IP 192.168.1.4.42988 > 18.229.250.79.443: Flags [P.], seq 3574341808:3574341864, ack
 1572886782, win 501, options [nop,nop,TS val 4224941699 ecr 1334653726], length 56
```

Figura 9 – Exemplo de uso do utilitário tcpdump. São coletados os pacotes TCP da interface de entrada "enp2s0", informações de IP e porta da fonte e do destino podem ser observadas.

O programa tc, abreviação de *Traffic Control* , tem a finalidade de gerenciar e monitorar o escalonador de filas de pacotes do Linux, funcionalidade essencial para o framework MINITY ter a capacidade de realizar medições em filas de pacotes de uma interface específica de hospedeiro. A Figura 10 apresenta o *script* utilizado para recuperar a informação da quantidade de bytes dos pacotes que estão na fila da interface especificada, a informação encontrando-se no campo *backlog*.

```
(base) root@rodrigoluna-Inspiron-7472:~# tc -s -d qdisc show dev enp2s0
qdisc fq_codel 0: root refcnt 2 limit 10240p flows 1024 quantum 1514 target 5.0ms interval 100.0ms m
emory_limit 32Mb ecn
 Sent 502073911 bytes 3292967 pkt (dropped 0, overlimits 0 requeues 1446)
 backlog 0b 0p requeues 1446
  maxpacket 2956 drop_overlimit 0 new_flow_count 7603 ecn_mark 0
  new_flows_len 0 old_flows_len 0
```

Figura 10 – Exemplo do utilitário tc. Neste exemplo informações do buffer de entrada da interface enp2s0 são coletadas. A quantidade de dados que estão enfileirados são informados através dos bytes no campo backlog.

Por fim, o utilitário ss é utilizado para coletar informações de *sockets*. Através dele é possível se obter informações de variáveis utilizadas nos algoritmos TCP, como o *Round Trip Time*, a janela de congestionamento, e a quantidades de bytes reconhecidos pelo destinatário. Com uso do ss pode-se também extrair infomações específicas das versões do TCP, como as variáveis de controle do BBR. A Figura 11 apresenta as informações específicas do BBR que podem ser extraídas a partir de um *parser*, como a *bandwidth* "bw", *minimum* rtt "mrtt", *pacing gain* e outras variáveis.

```
(base) rodrigoluna@rodrigoluna-Inspiron-7472:~$ ss -tin
State                        Recv-Q                        Send-Q
          Local Address:Port                                              Peer Address:Port


ESTAB                        0                             0
            192.168.1.4:38636                                            34.73.232.153:443

         bbr wscale:7,7 rto:400 rtt:170.314/26.228 ato:40 mss:1408 pmtu:1500 rcvmss:1408 advmss
:1448 cwnd:23 bytes_acked:2544 bytes_received:11047 segs_out:38 segs_in:34 data_segs_out:14 dat
a_segs_in:25 bbr:(bw:431.0Kbps,mrtt:153.298,pacing_gain:2.88672,cwnd_gain:2.88672) send 1.5Mbps
 lastsnd:43908 lastrcv:40620 lastack:40620 pacing_rate 2.1Mbps delivery_rate 431.4Kbps app_limi
ted busy:1100ms rcv_space:14480 rcv_ssthresh:64088 minrtt:142.009
```

Figura 11 – Exemplo de uso do utilitário ss. Através do sufixo -tin"pode-se capturar os endereços IP e porta da origem e do destinatário. Além disso valores das variáveis do TCP também estão disponíveis. Neste exemplo observa-se que o sistema operacional estava operando com o TCP BBR e as variáveis de controle específicas como: mrtt, pacing gain e cwnd gain podem ser capturadas.

3.2.2.2 Classe *Edge*: Armazenamento de informações das conexões

O framework MINITY modela o seu ambiente, hospedeiros e conexões utilizando o conceito de grafo, no qual os hospedeiros possuem ligações com outros hospedeiros atra-

vés de uma aresta que é modelada pela classe ***Edge***. Nesta classe são armazenadas as informações da conexão entre os hospedeiros, atributos necessários para que o ambiente MINITY possa configurar a rede a ser simulada. Para representar as máquinas virtuais, é necessário que existam rótulos para identificá-las. As arestas utilizam as informações dos rótulos para discernir as configurações de conexão entre os hospedeiros, ou seja, para cada aresta existe um único par de rótulos representado pelas variáveis h1 e h2. Os valores da banda, latência e taxa de perda de pacote da conexão, parâmetros intrínsecos à conexão, são armazenados nesta classe, como apresentado na representação UML na Figura 12.

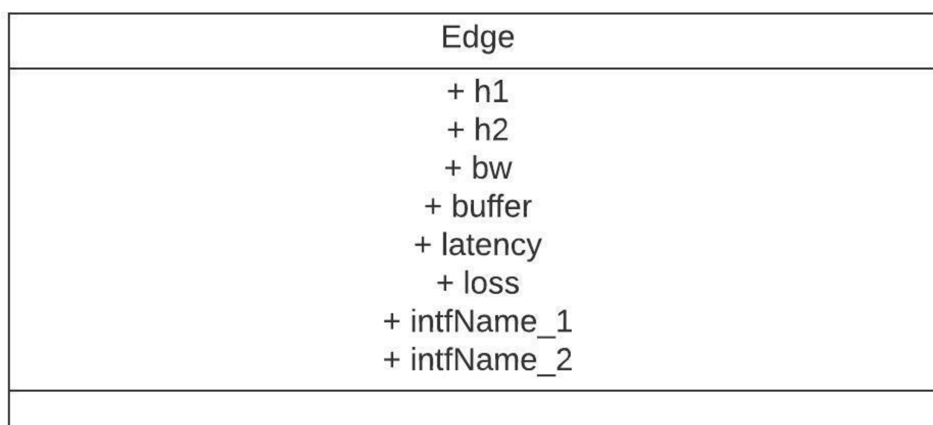


Figura 12 – Diagrama UML da Classe Edge.

As variáveis desta classe descrevem o controle de tráfego a ser realizado em uma conexão entre duas interfacese são utilizadas no sistema de qualidade de serviço das interfaces para o controle de tráfego dos pacotes. Deste modo, MINITY utiliza-se dos mecanismos de qualidade de serviço já disponíveis no kernel do sistema operacional Linux para realizar o controle de tráfego. Através do utilitário *Traffic Control* (ALEXEY N. KUZNETSOV, BERT HUBERT , 2001) pode-se acessar algoritmos já implementados no kernel do Linux para configurar a forma de como os pacotes recebidos pelo sistema operacional serão encaminhados para a rede. Na Figura 13 observa-se a esquematização do controle de tráfego utilizado pelo kernel do Linux. Ele é implementado através de dois mecanismos: 1) pelo policiamento dos pacotes na interface de entrada; e 2) a partir do enfileiramento dos pacotes na interface de saída. Desta maneira, pacotes podem ser atrasados, marcados, descartados ou priorizados.

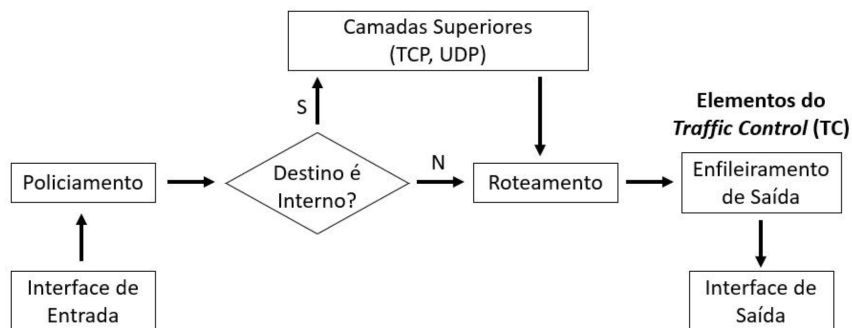


Figura 13 – Exemplificação do escalonamento da interface de rede do Linux. O pacote é recepcionado pela interface de entrada e são policiados, assim, pacotes indesejáveis são descartados. Caso o pacote seja de destino interno, será entregue para as camadas superiores. Senão, o pacote é enviado para o roteamento ou é oriundo de camadas superiores, assim, é enfileirado na saída, por onde podem ser atrasados, marcados, descartados ou priorizados, e enviados para a interface de saída. Imagem extraída de: (Edgar Jamhour, 2015)

Vale ressaltar que as políticas da qualidade de serviço são criadas de forma independente em cada interface de rede disponível no equipamento. Assim, o policiamento é realizado para os pacotes recebidos em uma certa interface e o controle de tráfego para os pacotes que são enviados pela interface.

A Figura 14 ilustra uma estrutura típica para o enfileiramento na interface de saída. Os elementos definidos pelo utilitário tc (ALEXEY N. KUZNETSOV, BERT HUBERT , 2001) são:

- *Qdisc*: abreviação de *Queuing discipline* e que corresponde aos algoritmos que controlam o enfileiramento e o envio de pacotes pela interface de saída;

- *Classes*: representam entidades de classificação de pacotes;

- *Filters*: são utilizados para policiar e classificar os pacotes e atribuí-los as suas respectivas classes;

- *Policers*: utilizados para evitar que o tráfego associado a cada filtro ultrapasse limites já definidos.

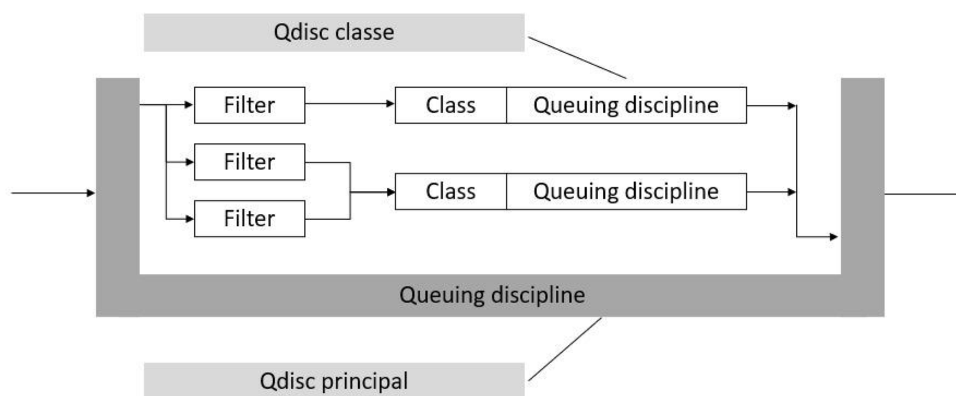


Figura 14 – A disciplina principal é chamada de *root* no comando tc. Além desta fila podem existir outras disciplinas pertencentes a classes de tráfego. A patir da distinção do tráfego de pacotes em uma interface, pode-se aplicar o controle de tráfego diferenciado para cada tipo de pacote que transita pela interface. Imagem extraída de: (Edgar Jamhour, 2015)

Observe que as *qdisc* são utilizadas em duas situações, uma principa, obrigatória, e outras associadas às classes. A *qdisc* principal define a regra geral da qualidade de serviço implementada pelo utilitário tc e também define a classe do pacote para o envio. Já as *qdiscs* das classes implementam o serviço diferenciado entre os tipos de pacotes. A quantidade de classes é variável e depende da quantidade de tratamentos diferenciados implementados pelo usuário através do tc. Os filtros são classificadores de pacotes com a função de encaminhar os pacotes para as respectivas classes para enfileiramento. Por exemplo, pacotes de dados são encaminhados para uma classe e pacotes VoIP para outra.

Para a identificação de uma *qdisc* utiliza-se um valor denominado handle, que possui o formato N:0, onde N é um inteiro qualquer. Ademais, as *qdiscs* são dispostas hierarquicamente em uma interface específica, como visto na Figura 15. A $qdisc_1$ root, *qdisc* principal, possui handle 1:. Já a $qdisc_2$ com handler 2: é uma disciplina filha do root, a $qdisc_n$ é filha da $qdisc_{n-1}$, como as outras que se seguem.

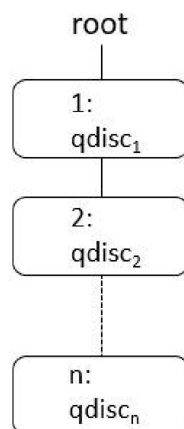


Figura 15 – Hierarquia das qdics utilizadas pelo utilitário tc. A qdisc_1 *root* é a principal. As *qdisc* subsequentes são filhas das anteriores. Imagem extraída de: (University of South California, 2019)

Para a criação de classes é necessário utilizar o atributo *parent* com o handle da *qdisc*

do pai para indicar que a classe é filha, além de utilizar o atributo *classid*, no seguinte formato "handle do pai:código da classe".

Na Figura 16 observa-se a esquematização da criação das classes filhas a partir da classe principal, além de demonstrar como os handles são definidos para cada uma das *qdiscs* apresentadas. Um pacote pode, então, ser redirecionado - através de filtros em cada um dos nós da árvore de representação - para a seguinte cadeia: 1: $>$ 1:1 $>$ 1:12 $>$ 12: $>$ 12:2, desta maneira, o pacote será enfileirado na classe 12:2. Filtros podem redirecionar pacotes para qualquer classe. Em MINITY não se utilizou filtros de pacotes por não haver necessidade de distinção no controle de tráfego neste trabalho. Uma consulta a (HUBERT et al., 2002) pode ser realizada para mais informações a respeito de como utilizar filtros, classes e policiamento.

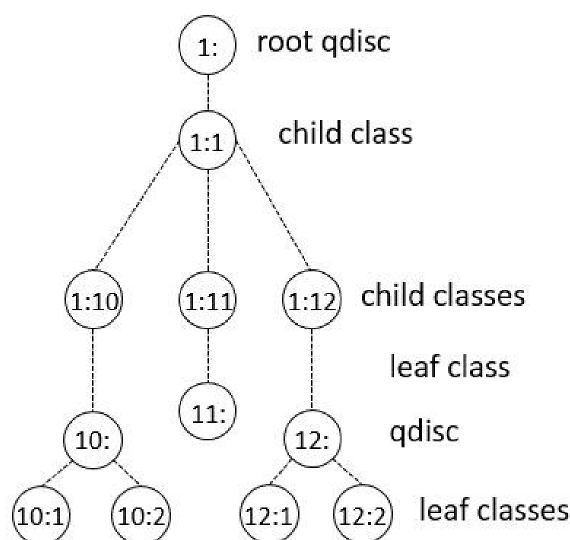


Figura 16 – Exemplificação da árvore de hierarquia das qdiscs disponíveis para o controle de tráfego no sistema operacional Linux, através do utilitário tc. A qdisc com o handle 1: é a principal. Já as qdiscs subsequentes são suas filhas e filhas das classes. Observe a composição do handle das classes filhas, são compostos sempre pelo formato (handle do pai):(código da classe). Imagem adaptada de (HUBERT et al., 2002)

Para exemplificar a utilização do tc para a construção da qualidade de serviço em uma interface, será criada uma qdisc principal com duas classes, uma com controle de taxa de envio de 500 Kbit/s e outra com 300 Kbit/s. Para a classe principal utiliza-se o seguinte código: **tc qdisc add dev eth0 root handle 1:0 htb**, em que htb é o algoritmo utilizado para o controle da taxa de envio de um pacote. Perceba que para a criação da disciplina em uma interface é necessário utilizar o atributo "add dev"precedido do nome da interface que se deseja realizar a operação. Em seguida, é indicada qual disciplina será criada, um *root* ou *parent*, sucedido pelo handle e do algoritmo a ser utilizado. Já para a adição de duas classes filhas é necessário utilizar a diretiva "parent"ao invés de "root", sucedido do handle da qdisc pai. Assim, para criar a primeira classe utiliza-se o seguinte código: **tc class add dev eth0 parent 1:0 classid 1:1 htb rate 500 Kbit**, observe que após a indicação da classe ser filha, utiliza-se o atributo "classid" sucedido do

formato "handle do pai":"handle da classe" e em seguida os atributos a serem utilizados pela diferenciação de serviço. Já para a segunda classe utiliza-se o código **tc class add dev eth0 parent 1:0 classid 1:2 htb rate 300 Kbit**. Observa-se que a diferença entre as classes se encontra na definição do nome da classe e na taxa de envio dos pacotes de cada classe. É importante ressaltar que não pode haver políticas implementadas em uma mesma interface com classid ou handle repetidos.

As unidades de medidas utilizadas pelo tc para configuração dos parâmetros podem ser consultadas em (HUNBERT, 2001). O parâmetro Kbit, utilizado nos exemplos, adota a unidade de kilobit por segundo.

Para o controle de tráfego no framework MINITY utilizou-se duas *qdiscs*. A *qdisc* principal, obrigatória, responsável pelo controle de banda, e uma *qdisc* filha para a simulação do atraso, variação do atraso, perda de pacotes e enfileiramento de pacotes. Na *qdisc* principal foi utilizado o algoritmo *Token Buffer Filter* (tbf) para controlar o limite da banda e a quantidade máxima de bytes recebidos por rajadas em uma interface. Na Figura 17 encontra-se o funcionamento do algoritmo que consiste na geração de tokens na velocidade desejada. Após o recebimento de um pacote, é averiguado se a fila de armazenamento de pacotes não enviados está lotada. Caso a fila esteja lotada, o pacote é descartado, caso ao contrário, ele é enfileirado e aguarda a liberação de um token para ser transmitido. Os parâmetros deste algoritmo são: 1) *limit* ou *latency*: indica a quantidade de pacotes que a fila do algoritmo suportará ou tempo máximo que um pacote irá permanecer na espera de um token; 2) *burst*: indica a quantidade de bytes em que tokens podem estar disponíveis instanteamente para garantir que pacotes recebidos em rajadas não sejam descartados; 3) *rate*: indica a taxa de banda que o algoritmo irá simular através da emissão dos tokens para os pacotes. Em MINITY a velocidade que o algoritmo irá implementar é o valor descrito na variável *bw* da classe Edge, encontrada na Figura 12, já o valor de *burst* é fixado em 15000 bytes.

Uma segunda *qdisc* com o *NetEm* (STEPHEN HEMMINGER, 2005) é utilizada para a simulação da latência, a probabilidade de perda de pacotes e a variação de latência, além de também ser responsável por armazenar pacotes excedentes. Os atributos da classe Edge pertencentes a esta *qdisc* são: 1) *latency*: indica o tempo de atraso que o pacote irá sofrer na qdisc do NetEm; 2) *loss*: indica a probabilidade de um pacote ser descartado no enfileiramento da saída; 3) *buffer*: indica a quantidade de pacotes que poderão ser armazenados na qdisc do NetEm.

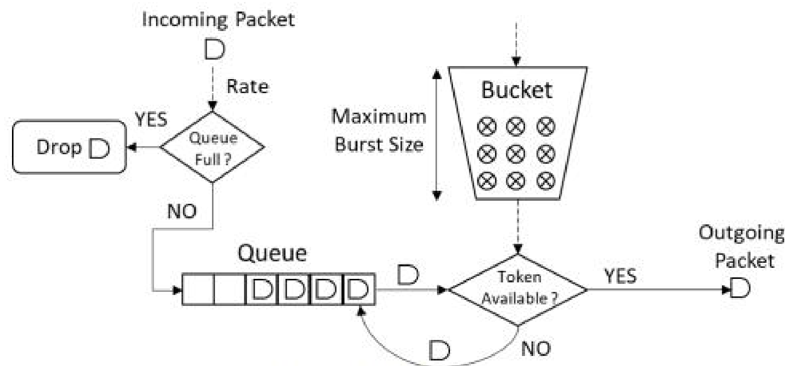


Figura 17 – Esquematização do algoritmo Token Bucket Filter para o controle de banda em uma disciplina de enfileiramento. O pacote recebido é enfileirado caso a fila não esteja cheia, caso ao contrário, é descartado. Os tokens são criados na velocidade em que se deseja transmitir o pacote, assim, quando um pacote só será transmitido caso a quantidade de tokens para sua transmissão esteja disponível. A fim de garantir que pacotes recebidos em rajadas não sejam descartados, o algoritmo define a quantidade de tokens que serão criados instanteamente. Imagem extraída de: (University of South California, 2019)

Na Figura 18 encontra-se um exemplo de uma interface do framework MINITY. Observa-se que a *qdisc* com o handle 5 root com o algoritmo *Token Buffer Filter* limita a taxa de envio em 100 Mbit/s, com rajada máxima de 15000 bytes, além do tempo máximo de enfileiramento em 1.2 ms. Já a *qdisc* com o handle 10 é filha da qdisc root e possui uma fila com capacidade de armazenar 1000 pacotes. É possível notar que não foram introduzidos atrasos no envio e nem perda de pacotes e estas são as configurações padrão implementadas em MINITY. Entretanto, as configurações podem ser alteradas pelo usuário através da interface interativa de linha de comando, como apresentado na Figura 5. Desta forma, qualquer configuração desejada pode ser adicionada, removida ou alterada pela interface.

```
(base) root@rodrigoluna-Inspiron-7472:~/Área de Trabalho/UFRJ/TCC# tc qdisc show dev sw2-sw3
qdisc tbf 5: root refcnt 2 rate 100Mbit burst 15000b lat 1.2ms
qdisc netem 10: parent 5:1 limit 1500
```

Figura 18 – Exemplo da configuração padrão de uma interface específica do framework MINITY.

3.2.2.3 Classe *Network*: Configuração dos parâmetros de rede

A flexibilidade para o usuário criar uma topologia genérica é a principal característica fornecida pelo framework MINITY. Dessa forma, é importante que exista uma interface para que o usuário possa implementar experimentos diversos através de alterações em um arquivo de configuração, de forma simples, prática e totalmente isenta de código de programação. Para que isso ocorra, é necessário que a classe ***Network*** possua um objeto da classe ***Topo***, da API MININET (BRANDON HELLER, 2013), para armazenar uma estrutura de dados com as informações dos nós e arestas da rede a ser simulada. Sendo assim, a classe ***Network*** recebe em seu construtor de classe um objeto da classe

Topo, contendo as informações gerais da rede, para que se possa configurá-la através dos métodos: *trafficShapping*, *callSniffer*, *configHosts*, *configSwitches* e *configRouter*, que podem ser encontrados na representação em UML na Figura 19.

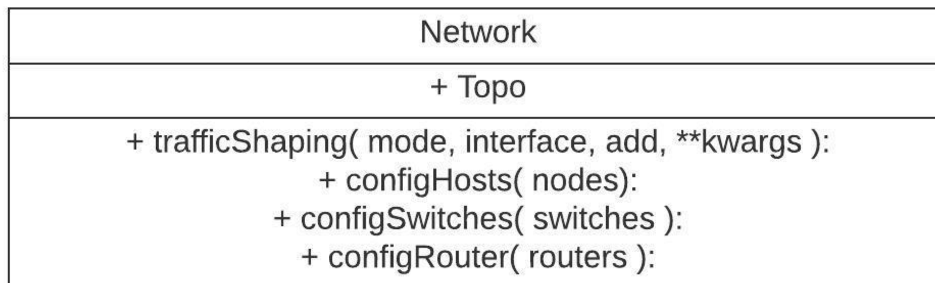


Figura 19 – Diagrama UML da Classe Network.

Os métodos da classe ***Network***, além de configurar as interfaces de rede dos hospedeiros, também ajusta o sistema de gerenciamento de arquivos que será empregado para o armazenamento dos dados coletados. Os resultados sempre serão adicionados nos diretórios *results/rótulo do hospedeiro*. Ademais, os métodos *configSwitches* e *configRouter* configuram as pastas de coleta de dados dos nós *switches* e roteadores nos mesmos diretórios dos hospedeiros nos quais os resultados também são adicionados ao diretório *results/rótulo*.

O método *configHosts* tem como objetivo configurar a máquina virtual de acordo com os parâmetros definidos pelo usuário, e através dele, é possível configurar a latência na rede - sendo possível simular atraso na entrega do pacote -, a taxa de perda de pacote e a variação de latência através do escalonamento de filas utilizando os pacotes do Linux, o *NetEm* (STEPHEN HEMMINGER, 2005) e o *tc* (ALEXEY N. KUZNETSOV, BERT HUBERT , 2001).

A simulação da latência, perda de pacote e a variação da latência no framework MINITY são de responsabilidade das interfaces de saída dos hospedeiros. No caso do atraso de propagação ida e volta, nos experimentos realizados por este trabalho, optou-se por deixar a cargo da interface de saída do hospedeiro cliente a simulação deste atraso. Entretanto, nada impede o usuário de adicionar latência na interface de saída de cada hospedeiro, seja cliente ou servidor, ou das interfaces dos comutadores, para atribuir o atraso de propagação por enlace. Esta simulação do atraso de propagação é necessária devido à dificuldade de simular uma métrica que possua características relacionadas a distâncias físicas e aos tipos dos meios de propagação no ambiente de emulação lógico criado pelo MININET.

Nos protocolos TCP, amostras do tempo de propagação de ida e volta, RTT (*Round Trip Time*), são obtidas nas chegadas das confirmações de segmentos, quando se mede o tempo entre o envio do segmento e o recebimento da confirmação. O RTT médio estimado a partir destas amostras está então considerando todos os atrasos ao longo da

conexão TCP, como o tempo de propagação, o tempo de enfileiramento e transmissão, atrasos nos roteadores e comutadores intermediários, além do tempo de processamento nos hospedeiros.

Nas topologias de Internet, o RTT é da ordem de dezenas ou centenas de milissegundos, dependendo das distâncias e número de hops envolvidos. Quando não há formação de fila em gargalos, temos valores mínimos de RTT. Como não há vantagem em adicionar complexidade com a inserção de vários hops sendo que apenas um gargalo é normalmente observado e simulado numa conexão TCP, optou-se por configurar apenas um único valor de latência equivalente ao RTT mínimo alvo na interface de saída do cliente. Assim, o envio de dados do servidor para o cliente experimentará 0 de latência, enquanto na direção oposta de tráfego, o envio de pacotes de confirmação do cliente para o servidor, é penalizado com a latência equivalente ao RTT mínimo. O protocolo TCP irá então medir um RTT que será esta latência mínima acrescida do tempo de atraso na fila no gargalo e dos tempos de retransmissão dos pacotes pelos dispositivos da emulação Teremos então uma emulação equivalente a uma topologia com vários hops. Vale observar que nenhuma outra latência é adicionada nas filas de saída dos dispositivos.

A simulação na latência é realizada através do escalonamento de pacotes da interface de rede do sistema operacional Linux, encontrado na Figura 13. Através da interface de entrada os pacotes são recepcionados e redirecionados para a camada superior ou para o roteamento do pacote. Antes deste encaminhamento, os pacotes são policiados na entrada e é através dele que pacotes indesejáveis são descartados. No envio da confirmação do recebimento pacote pelo hospedeiro cliente, o pacote ACK é enfileirado na saída antes de ser enviado para o servidor, quando a latência é adicionada.

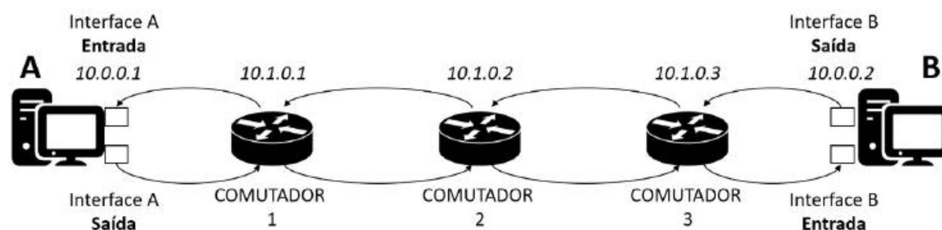


Figura 20 – Exemplificação das interfaces de entrada e saída de uma transmissão.

Para exemplificar a construção de uma topologia com determinado RTT alvo, toma-se a Figura 20 como exemplo. Nesta esquematização, há dois hospedeiros, um servidor e um cliente - sendo representados por A e B, respectivamente - e a presença de três comutadores. Trata-se de uma sub-rede local com endereçamento 10.0.0.0/8. Observa-se que os links entre os equipamentos são bidirecionais e os hospedeiros A e B possuem interfaces distintas de entrada e saída de pacotes. A configuração da latência, perda e a variação de latência são configuradas no hospedeiro B em sua interface de saída, que segue o escalonamento de pacotes exemplificado na Figura 13. Desta maneira, qualquer

envio de pacotes de A para B não sofre qualquer atraso, sendo tranmsitidos de imediato no enlace entre A e o comutador 1. Já o envio de pacotes de B para A, ao serem recepcionados pela interface de saída B, sofrem as condições esperadas de perda, latência e variação de latência. Os comutadores são do tipo *store-and-foward*, assim cada quadro de enlace só é transmitido após o recebimento integral do quadro. Nesta topologia, não haverá conflito na comutação dos pacotes e nem ocorrência de fila, se todos os enlaces forem de mesma velocidade.

Devido ao interesse motivador de usar MINITY para aferir o comportamento dos protocolos TCP e pelo fato da API do emulador de redes MININET não possuir uma forma simples da adição de roteadores, apenas o uso de comutadores foi contemplado nesta versão inicial da ferramenta. Como não há o objetivo de analisar os algoritmos de roteamento, a adição de roteadores amentaria a complexidade para a criação de um experimento, e, consequentemente, também a análise de um protocolo TCP. Entretanto, caso seja necessário, esta funcionalidade pode ser adicionada através da emulação de um roteador a partir de um hospedeiro, com a configuração do Linux para efetuar roteamento.

O método *configHosts* também verifica a disponibilidade do algoritmo TCP no sistema operacional vigente e, a partir do comando *sysctl* (GEORGE STAIKOS, 1999), define o algoritmo a ser utilizado na respectiva máquina virtual. Por fim, caso o hospedeiro possua um servidor FTP, é nesse método que será realizado a chamada da função *configServer* disponível na classe ***Node***.

O Método *trafficShapping* auxilia a classe ***Network*** na criação do comando a ser utilizado pela ferramenta *tc*. Dado o conjunto de informações a respeito de uma conexão, é implementado na interface destinatária o conjunto de ações que o pacote experimentará. Esse comando é então executado pela máquina destinatária do pacote a fim de configurar o sistema de enfileiramento dos pacotes.

Por fim, o método *callSniffer* tem a atribuição de chamar as funções do objeto para ativar a coleta de dados. Essas funções são definidas na classe ***Node***: `run_Sniffer`, `run_bufferScript` e `run_ssScript`

Portanto, a classe ***Network*** é imprescindível para a configuração de todo o ambiente MINITY e da concentração das ações a serem realizadas. É através dela que são iniciadas e configuradas as máquinas virtuais, o ambiente de captura de dados, os *scripts* que realizam a captura de dados e o controle de enfilamento de pacotes. Para realizar qualquer alteração ou adicionar novas funções em qualquer classe do ambiente, é necessário inicializá-lo a partir desta classe.

3.2.2.4 Classe *Minity*

A classe ***Minity*** é o ponto de entrada para o framework MINITY. É nesta classe que os arquivos de configuração e de ajuste do experimento são lidos e configurados. Através da Figura 6, pode-se observar o fluxo de execução do framework MINITY.

O construtor da topologia é o responsável pela leitura dos arquivos de configuração e também por inicializar a estrutura. Diversas funcionalidades foram acrescentadas a essa estrutura para que o usuário possua a capacidade de realizar testes no ambiente em que se irá realizar o experimento real. Através de um comando de interface disponível na API da MININET (BRANDON HELLER, 2013), o usuário poderá verificar se todas as conexões criadas estão de acordo com o arquivo de configuração, a fim de garantir que não exista nenhum problema na topologia. Através de dois arquivos de configuração é possível realizar uma manutenção ou alteração manual no experimento.

Foi criada no MINITY uma funcionalidade para realizar testes automatizados sobre toda a topologia com a intenção de checar se as conexões estão com os parâmetros corretamente definidos e averiguar o quanto de banda uma máquina virtual consegue transmitir para outra máquina virtual. Para viabilizar esta função, a ferramenta para teste de performance *iperf* (MARK G.,ALEX W., 2008) foi incorporada ao ambiente. A funcionalidade de testes automatizados foi adicionada na API da MININET através do método *iperf()*, que recebe como parâmetro o protocolo da camada de transporte pelo atributo *l4Type*. Para ativar essa funcionalidade no framework MINITY é necessário executar o comando run(iperf=True), como pode ser visto na Figura 24a. Como o motivador deste trabalho é realizar experimentos do protocolo TCP, o valor padrão definido para *l4Type* será 'TCP', podendo o valor ser alterado para 'UDP', caso o usuário deseje. O ponto de entrada dessa classe é o método *run()* e é através dele que se inicia o sistema como um todo.

No arquivo de configuração do experimento existe um parâmetro que define o número de rodadas do experimento para que sejam gerados dados suficientes a fim de gerar resultados estatisticamente confiáveis. É importante observar que a cada rodada coleta-se todo o comportamento transiente do tráfego TCP, sem a existência de gargalos devido a tráfego de fundo, sem a concorrência de outras conexões nos hospedeiros envolvidos e com as filas do experimento se incializando completamente vazias. Desta maneira, o estado de partida de toda rodada do experimento será sempre o mesmo.

3.2.2.5 Classe Handler

Na Figura 6 foi definido um arquivo de configuração chamado *experimento.json* e é através dele que a classe ***Handler*** configura todos os parâmetros do experimento.

A classe ***Handler*** gerencia o sistema de acordo com uma linha do tempo discretizada em segundos, a cada instante de tempo sendo executados os eventos determinados para cada hospedeiro. Optou-se pela discretização do tempo em segundos para facilitar o usuário na configuração dos disparos dos eventos e também para simplificar o desenvolvimento do módulo Handler, visto que, gerenciar eventos em tempo contínuo ocasionaria em uma maior complexidade para o sistema de gerenciamento de eventos do MINITY.

Diversas ações podem ser efetuadas durante o experimento. Na Figura 21 pode-se observar todas as possíveis ações que um hospedeiro está habilitado para executar. As

params_exp_1.json

```
[
    {
        "type":"config",
        "n_rodadas":1,
        "tempo_rodada":100
    },
    {
        "time":1,
        "node":"h4",
        "type":"start",
        "ip":"10.0.0.1",
        "filename":"big_file.zip"
    },
    {
        "type":"bw",
        "time":20,
        "node":"sw2",
        "intfName":"sw2-sw3",
        "value":20
    },
    {
        "type":"bw",
        "time":40,
        "node":"sw2",
        "intfName":"sw2-sw3",
        "value":10
    },
    {
        "type":"rtt",
        "time":55,
        "node":"h4",
        "intfName":"sw3-h4",
        "value":"120ms"
    },
    {
        "type":"rtt",
        "time":75,
        "node":"h4",
        "intfName":"sw3-h4",
        "value":"40ms"
    }
]
```

Figura 21 – Formato Experimento JSON. Neste arquivo configura-se o experimento que será executado pelo framework MINITY. Através dele, todas as ações de alterações de banda, rtt e outras ações serão configurados.

ações são configuradas através de um arquivo json, composto por chaves com informações referentes aos comandos e os seus respectivos valores.

O comando *type* é responsável por indicar qual será a alteração dessa instância no momento da execução, podendo assumir os valores: *config*, *start*, *bw* e *rtt*. No modo *config* é informado à classe ***Handler*** parâmetros indicando a quantidade de rodadas que o experimento irá possuir e o tempo total de execução de cada rodada em segundos, através dos campos *n_rodadas* e *tempo_rodada*. No modo *start* é informado à classe ***Handler*** exatamente quando um hospedeiro irá iniciar a transferência de algum arquivo e os campos para esse formato são: *time*, *node*, *ip* e *filename*, sendo, respectivamente, o tempo de quando a ação irá ocorrer, o nome do hospedeiro que irá executar a ação, o endereço IP do servidor FTP que possui o arquivo a ser requisitado e o nome do arquivo a ser transferido. Os valores *bw* e *rtt* configuram, respectivamente, a taxa de banda que o hospedeiro tem em sua interface e o RTT (*Round Trip Time*). Seus campos são: *time*,

node, *intfName* e *value* , que indicam, respectivamente, o tempo, em segundos, que essa ação será executada, o nome do hospedeiro, o nome da interface que irá sofrer a ação de alteração de banda ou RTT, e o valor a ser ajustado. Nesse caso, quando o *type* recebe o comando *bw*, o valor representa a taxa de banda em Mbit/s e, quando o *type* recebe o comando *rtt*, o valor representa o RTT em milissegundos.

3.2.2.5.1 Arquivos de Configurações

No projeto de desenvolvimento do framework MINITY, a usabilidade do sistema não depende de nenhuma alteração no código fonte. Para que isso fosse possível, foi necessário esquematizar um sistema de arquivos para a configuração do ambiente e do experimento. Sendo assim, foi escolhido o modelo de arquivo JSON (BRAY, 2017). Trata-se de um formato de troca de informações e dados entre sistemas legíveis para o usuário. Um exemplo de um modelo JSON pode ser encontrado na Figura 22. Os campos são sempre definidos como um par chave e valor. No ambiente MINITY, como os parâmetros que o usuário pode alterar são pré-definidos, esse modelo se adequa perfeitamente ao ambiente desenvolvido. A função do python utilizada foi a *json*. Para carregar a informação em uma variável de simples acesso é necessário realizar a chamada através do método *load*.

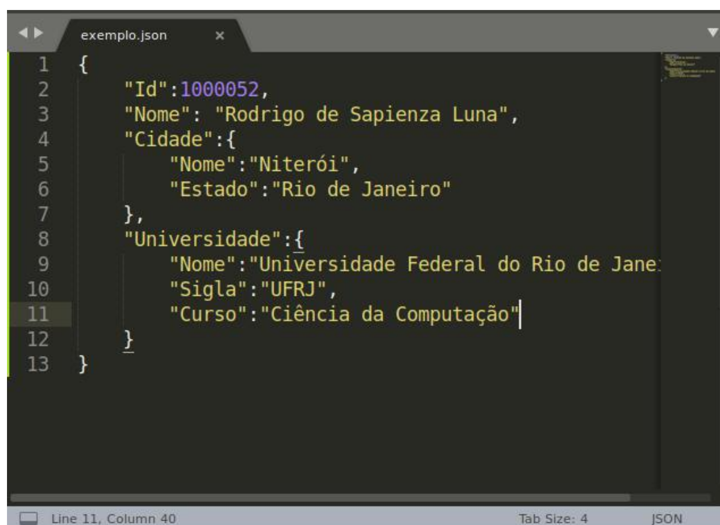


Figura 22 – Exemplo de arquivo no formato json

Como pode ser observado na Figura 6, o construtor da topologia recebe como entrada dois arquivos de configuração, *topologia.json* e o *experimento.json*, que podem ser encontrados no diretório **minity/config**.

Nas subseções seguintes serão descritos em detalhes a formatação dos arquivos e seus parâmetros para auxiliar os futuros usuários.

3.2.2.6 Rotina de Configuração

As informações de configuração relacionadas ao ambiente MINITY são mantidas em um arquivo de formato JSON (vide Figura 22). É neste arquivo que os parâmetros das classes ***Node*** e ***Edge*** são definidos e podem ser alterados pelo usuário. Cada uma dessas estruturas possui um escopo pré-definido para facilitar o usuário na configuração de parâmetros, sendo necessário apenas copiar a estrutura para a construção de um novo objeto, seja da classe ***Node*** ou ***Edge***.

```
{
        "type":"HOST",
        "name":"h1",
        "transport_protocol":"bbr",
        "queue":{
            "protocol":"",
            "latency":"100ms",
            "length":"",
            "jitter":"10",
            "variation":"5",
            "loss":""
        }
        ,
        "sniffer":
            {
                "queue":{

                    "status":1,
                    "delta_t":0.01,
                    "intf":"h1-eth1"
                },
                "traffic":{
                    "status":1,
                    "intf":"h1-eth1"
                },
                "socket":{
                    "status":1,
                    "delta_t":0.02
                }

            },
        "ip":"10.0.0.1",
        "ftp_server":1
    },
```

Figura 23 – Exemplo de configuração de um Node. O Hospedeiro "h1"é um cliente e possui latência de 100ms. Ademais informações da fila de pacotes da interface de entrada serão coletados.

A Figura 23 apresenta uma estrutura de um objeto *Node*, onde o usuário define os parâmetros de configuração geral do hospedeiro, da fila de pacotes e dos parâmetros da classe ***Sniffer***.

O parâmetro *type* indica o tipo do objeto para que o ambiente possa identificar os campos que serão lidos, e, assim, criar o objeto correspondente à tipagem definida. O parâmetro *name* é extremamente importante e o usuário deve configurá-lo com cautela,

porque o ambiente MINITY o utiliza para identificar as ligações e ações do sistema.

O campo *transport_protocol* é usado para informar ao sistema qual algoritmo o hospedeiro utilizará. Como neste trabalho queremos analisar o comportamento do protocolo BBR e de outros protocolos TCP, o exemplo demonstrado pela Figura 23 utiliza este protocolo. No entanto, pode-se utilizar qualquer algoritmo que esteja disponível no sistema operacional do usuário.

Os parâmetros da *queue* do sistema são configurados através desse arquivo de configuração. Esses parâmetros configuram as filas disponibilizadas pelo sistema operacional Linux para a camada 3 do modelo TCP/IP, ou seja, os sistemas finais. Ademais, as filas configuradas são para o envio de pacotes, ou seja, filas de saída e caso não sejam configuradas pelo usuário, permanece implementado as configurações padrões do sistema operacional Linux. O protocolo a ser utilizado para o gerenciamento da fila é modificado através da variável *protocol*, sendo que, no exemplo, é utilizado o protocolo fq codel (HOEILAND-JOERGENSEN et al., 2018). Além do protocolo, pode-se definir outras características da fila: o tamanho da fila, sendo o padrão 1000 pacotes , quando não possuir nenhum valor; simular atraso na entrega do pacote; o *jitter* (DEMICHELIS; CHIMENTO, 2002), isto é, a variação dessa entrega à aplicação destinatária, simulado em sua própria interface; e a porcentagem de perda de pacote, que é a probabilidade de um pacote ser descartado na interface receptora.

Cada *host* necessita realizar coletas de dados em diferentes filas a fim de obter as métricas de interesse. Estas coletas são realizadas: 1) na fila de saída de pacotes do sistema operacional, a partir de amostragens em intervalos de tempo definido pelo usuário; 2) na interface de rede do sistema operacional, onde são coletados todos os pacotes que transitam para se obter a taxa de transferência média da conexão; e 3) no *socket* da comunicação TCP são capturados por meio de uma amostragem realizada em intervalo de tempo definido pelo usuário, para obter detalhes das variáveis de controle do algoritmo em uso. Sendo sua execução e implementação de responsabilidade da classe ***Sniffer***, as três coletas possuem configurações específicas, no entanto, com parâmetros em comum, como o *status*, *delta_t* e o *intf* que são, respectivamente, uma variável de controle booleana - que indica se o hospedeiro irá ou não utilizar essa funcionalidade -, tempo entre as amostragens, e o nome da interface que terá dados coletados.

Como pode ser visto na Figura 23, o hospedeiro "h1"está realizando uma amostragem na fila de saída da interface a cada 10 ms. Em seguida, são coletados os pacotes que transitam pela interface de nome "h1-eth1"no tráfego. Por fim, as informações do socket da comunicação TCP são coletadas por uma amostragem a cada 20 ms.

3.2.3 Bloco *Analyzer*

Esta seção descreve a organização e o funcionamento do bloco responsável pelo tratamento dos dados brutos obtidos pelo experimento e a sua modularização para a extração

das métricas do experimento. O bloco *Analyzer*, cujo diagrama de funcionalidade é mostrado na Figura 7, contém três módulos: *Extrator*, *Coletor* e *Plotagem*. Inicialmente, após a execução do experimento, o módulo *Extrator* identifica os arquivos que contém os dados obtidos pelo bloco *Handler* e empreende o tratamento dos dados brutos. Após isso, o módulo *Coletor* executará rotinas que irão, a partir dos dados brutos selecionados pelo módulo *Extrator*, extrair informações do experimento e armazenar as métricas em tabelas. A partir destas tabelas, o módulo *Plotagem* realiza a leitura dos dados, gera os gráficos das métricas e os disponibilizam para o usuário. Os três módulos são explicados com mais detalhes abaixo.

3.2.3.1 Módulo *Extrator*

A atribuição deste módulo é efetuar uma varredura pelas pastas do MINITY a fim de identificar, coletar e unificar os arquivos contendo os dados brutos gerados pelo experimento. Após isso, este módulo executa as funções fornecidas pelo módulo *Coletor* que, após o tratamento dos dados brutos, devolve para o módulo *Extrator* os dados tratados que são, então, armazenados em tabelas.

A função responsável por este módulo é a *extract*, que possui como parâmetro o delta_t, tendo como valor padrão 0.173 em segundos, obtido experimentalmente, para que as métricas do módulo *Coletor* obtenham estimações consistentes e para que sejam capazes de captar, de maneira sutil, as alterações efetuadas no experimento.

A variável delta_t da função extract foi definida especificamente para obter a taxa de envio entre dois hospedeiros. Este valor é repassado para a função sendingRate, implementada no módulo *Coletor*, que irá obter a estimativa da taxa de envio a partir da divisão do tempo total do experimento, em blocos de tamanho delta_t, avaliando a quantidade de bits, em Mbit/s, que são transmitidos a cada bloco. Ademais, este valor é próximo aos valores dos RTTs utilizados neste trabalho, entre 40 ms e 120 ms. Deste modo, torna-se possível capturar as mudanças de fase do TCP BBR durante os experimentos realizados.

Todos os resultados serão salvos em um único diretório do ambiente MINITY, "Minity/analyzer/tables/", com a extensão *.csv*, para que possa ser interpretado como tabela, de uma maneira mais simples e organizada.

3.2.3.2 Módulo *Coletor*

Esse módulo é responsável pela criação dos algoritmos responsáveis pela interpretação dos dados brutos. Ele é capaz de interpretar os resultados obtidos pelo experimento e realizar a estruturação do dado para que seja possível extrair as métricas de interesse. Nesse trabalho foram utilizadas três funções principais, que modularizam a informação: *sendingRate*, *bbrParser* e *queueParser*.

A função *sendingRate* tem como objetivo interpretar os dados coletados pela interface de rede e estimar a taxa de envio entre dois hospedeiros. A extensão *pcap* é utilizada pelo arquivo que contém informações dos dados transmitidos. Como a coleta de dados é feita por amostragem a cada intervalo de tempo delta_t pela classe ***Sniffer*** em uma interface com eventualmente inúmeros fluxos compartilhando o mesmo gargalo, é necessário que o algoritmo seja capaz de interpretar quais sãos os hospedeiros fonte e destino. Devido a múltiplas conexões estarem ativas no momento da coleta, também é preciso diferenciar uma conexão através de uma 4-tupla, composta por: ip da fonte, ip destino, porta fonte e porta destino.

Além disso, como o protocolo FTP utiliza duas portas, uma para a transferência de dados e outra para o controle da conexão, torna-se necessário contabilizar apenas o tráfego da conexão de dados. Após a identificação da conexão de dados, os bytes utilizados para a contagem são os dados do campo *data* do segmento TCP, ignorando os bytes do cabeçalho do protocolo IP e do TCP. Devido a taxa de envio ser uma estimativa de bits por segundo, a cada intervalo de tempo $[t_i, t_i + \Delta t]$ é calculado a soma dos bytes dos pacotes transmitidos, assim é possível estimar a taxa média de bit/segundo entre os hospedeiros. Ao fim da execução desta rotina, uma tabela é gerada, contendo uma linha para cada 4-tupla com informações das conexões de todo o experimento, como o tempo medido, a taxa de envio e um identificador para que seja possível diferenciar as diversas rodadas do experimento. Assim, é possível obter uma estimativa da quantidade de bytes que está sendo transmitida ao longo da conexão e capturar as alterações de tráfego que venham a ocorrer.

Já o método *bbrParser* é específico para se obter informações das variáveis internas do protocolo BRR, obtidos a partir de informações dos sockets que são implementados pelo istema operacional. Esses dados não estruturados são transformados em dados estruturados para que a tabela final possua informações das variáveis de controle, como o instante de tempo de captura.

Por fim, o método *queueParser* fornece a estruturação dos dados extraídos de uma interface de rede de entrada após uma amostragem de dados. Informações como: RTT, quantidade de bytes no buffer da fila da interface de entrada, a taxa de perda de pacotes, o limite de pacotes em bytes, o atraso e a quantidade de pacotes que pode ser recebido por rajada. Essas informações são extraídas e inseridas numa tabela final para auxiliar a visualização das métricas.

3.2.3.3 Módulo Plotagem

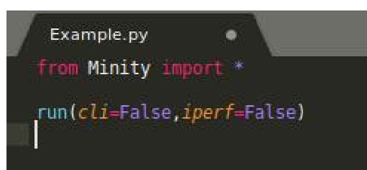
Este é o último módulo do bloco *Analyzer*, como apresentado na Figura 7. Ele é responsável pela geração de gráficos do framework MINITY, criado a partir dos dados obtidos das métricas disponibilizadas pelo módulo *Coletor*. As funções usam a biblioteca matplotlib (HUNTER, 2007) para a geração de gráficos e a biblioteca Pandas (TEAM,

2020) para a leitura da tabela. Os gráficos, após de serem gerados, são armazenados na pasta *results*.

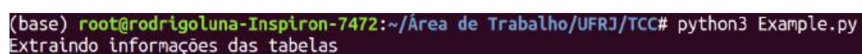
Dado que o motivador deste trabalho é o desenvolvimento de uma ferramenta que permita a análise do protocolo TCP BBR em diversos cenários, os gráficos foram definidos para exibir os valores das variáveis de controle deste protocolo e dos valores reais coletados pelo framework, para que assim, seja possível comparar os valores estimados internamente pelo protocolo com os valores reais da conexão, como, por exemplo, gráficos com os valores estimados da banda e do RTT medidos pelo BBR. Caso seja necessário a obtenção dos gráficos de variáveis de outros protocolos, estes gráficos terão que ser implementados. Já os gráficos referentes às variáveis da interface de rede do sistema operacional já estão implementados. A adição de gráficos pode ser efetuada através das tabelas geradas pelo módulo Coletor.

Para utilizar as funções deste módulo o usuário deve seguir uma das duas alternativas propostas por este trabalho. Uma possível execução é através do método *run()*, que pode ser utilizado após a importação das funções do framework MINITY. Através deste método todas as funções que geram gráficos são executadas. Este método é utilizado tanto para executar o experimento quanto para executar o módulo *Plotagem*, diferindo-se apenas pelo interpretador do python. Para o módulo *Plotagem* é imprescindível a versão python 3 ou superior.

Uma segunda alternativa é a chamada de funções de forma independente, através da importação dos métodos, como apresentado na Figura 24a, em que as funções do framework MINITY são importadas para o arquivo, e desta maneira podem ser executadas individualmente.



(a) Importação dos métodos para um arquivo Python.



(b) Execução

Figura 24 – Exemplo de como executar os métodos dos Gráficos do Módulo Coletor. Em (a) observa-se um código com a chamada da função run() para a execução de todos os gráficos do módulo Coletor; em (b) a sua execução a partir do Python 3 para a geração dos gráficos.

4 EXEMPLO DE USO DO MINITY

Nesta seção será exemplificada a criação de um experimento com a finalidade de preparar os futuros usuários para utilizarem o framework MINITY. Este experimento consiste de dois fluxos TCP BBR de dois servidores transferindo aqruivo para dois clientes distintos, através de três comutadores. Além disso, durante o experimento serão realizadas alterações nos parâmetros da rede, como a redução ou o aumento da banda disponível no gargalo e alterações nos RTTs.

4.1 TOPOLOGIA

A topologia de *dumbbell* será utilizada para exemplificar a modelagem da rede. Esta topologia é caracterizada por possuir N Hospedeiros servidores e N Hospedeiros clientes compartilhando um único enlace de gargalo através de três comutadores, como apresentado na Figura 25. No exemplo, a seguir, será utilizado como N = 2, com a finalidade de exemplificar e facilitar a compreensão do usuário no manuseio do framework.

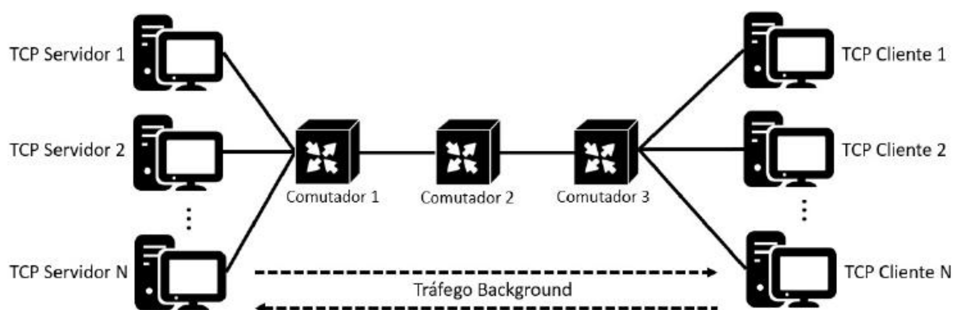


Figura 25 – Representação da topologia Dumbbell. Através dela, N hospedeiros servidor transmitem dados para N hospedeiros cliente através de três comutadores.

4.2 CONFIGURAÇÃO

Essa seção detalha, minuciosamente, os parâmetros de configuração do experimento utilizados por cada componente do framework MINITY. A utilização de três comutadores tem como objetivo garantir um gargalho entre as conexões, que será realizado pelo link entre o Comutador 2 e Comutador 3 da Figura 25. Além disso, pode-se observar que os Hospedeiros ou são servidores ou clientes, sendo representados, respectivamente, como TCP Servidor e TCP Cliente. Alguns dos campos das configurações são definidos para o ajuste de parâmetros, permitindo configurar o nome do hospedeiro, o tipo (servidor ou cliente), parâmetros da fila, parâmetros de coleta de dados da fila, do tráfego e do *socket*, o ip a ser utilizado e, por fim, um parâmetro para indicar se o servidor FTP está ativo ou não.

```
{
    "type":"HOST",
    "name":"h1",
    "transport_protocol":"bbr",
    "queue":{
        "protocol":"",
        "latency":"0ms",
        "length":"",
        "jitter":"",
        "variation":"",
        "loss":""
    }
    ,
    "sniffer":
        {
            "queue":{
                "status":0,
                "delta_t":0,
                "intf":""
            },
            "traffic":{
                "status":0,
                "intf":"",
            },
            "socket":{
                "status":1,
                "delta_t":0.002
            }

        },
    "ip":"10.0.0.1",
    "ftp_server":1
```


(a) Hospedeiro Servidor h1


```
{
    "type":"HOST",
    "name":"h2",
    "transport_protocol":"bbr",
    "queue":{
        "protocol":"",
        "latency":"0ms",
        "length":"",
        "jitter":"",
        "variation":"",
        "loss":""
    }
    ,
    "sniffer":
        {
            "queue":{

                "status":0,
                "delta_t":0,
                "intf":""
            },
            "traffic":{
                "status":0,
                "intf":"",
            },
            "socket":{
                "status":1,
                "delta_t":0.01
            }

        },
    "ip":"10.0.0.2",
    "ftp_server":1
},
```


(b) Hospedeiro Servidor h2


```
{
    "type":"HOST",
    "name":"h3",
    "transport_protocol":"bbr",
    "queue":{
        "protocol":"",
        "latency":"80ms",
        "length":"",
        "jitter":"",
        "variation":"",
        "loss":""
    }
    ,
    "sniffer":
        {
            "queue":{
                "status":1,
                "delta_t":0.001,
                "intf":"h3-eth4"
            },
            "traffic":{
                "status":0,
                "intf":""
            },
            "socket":{
                "status":0,
                "delta_t":0
            }

        },
    "ip":"10.0.0.3",
    "ftp_server":0
},
```


(c) Hospedeiro Cliente h3


```
{
    "type":"HOST",
    "name":"h4",
    "transport_protocol":"bbr",
    "queue":{
        "protocol":"",
        "latency":"40ms",
        "length":"",
        "jitter":"",
        "variation":"",
        "loss":""
    },
    "sniffer":
        {
            "queue":{
                "status":1,
                "delta_t":0.001,
                "intf":"h4-eth4"

            },
            "traffic":{
                "status":0,
                "intf":""
            },
            "socket":{
                "status":0,
                "delta_t":0
            }

        },
    "ip":"10.0.0.4",
    "ftp_server":0
},
```


(d) Hospedeiro Cliente h4

Figura 26 – Configuração dos Hospedeiros do experimento relatado. Em a) e b) representa a configuração, respectivamente, dos hospedeiros fontes h1 e h2. Já em c) e d) reporta a configuração dos hospedeiros destinos h3 e h4.

O *Round Trip Time* será ajustado nos Hospedeiros clientes. Como já mencionado em seção anterior, o RTT (o tempo de ida e volta entre origem e destino) é simulado pelo MINITY através da adição de um atraso fixo na fila de saída da interface de rede do

hospedeiro cliente. O RTT dos hospedeiros h3 e h4 será de 40 ms e 80 ms, respectivamente, como apresentado nas Figuras 26c e 26d. Os Hospedeiros clientes coletam informações de suas respectivas filas, já os Hospedeiros servidores coletam informações do *socket* para extrair os valores das váriaveis do protocolo TCP durante a transmissão de dados.


```
{
    "type":"SWITCH",
    "name":"sw1",
    "transport_protocol":"bbr",
    "queue":{
        "protocol":"",
        "latency":"",
        "length":"",
        "jitter":"",
        "variation":"",
        "loss":""
    },
    "sniffer":
        {
            "queue":{
                "status":1,
                "delta_t":0.01,
                "intf":"sw1-eth4"
            },
            "traffic":{
                "status":1,
                "intf":"sw1-eth4"
            },
            "socket":{
                "status":0,
                "delta_t":0
            }

        },
    "ip":"10.0.1.1",
    "ftp_server":0,
    "mac":"00:00:00:00:00:01"
},
```


(a) Comutador 1.


```
{
    "type":"SWITCH",
    "name":"sw2",
    "transport_protocol":"bbr",
    "queue":{
        "protocol":"",
        "latency":"",
        "length":"",
        "jitter":"",
        "variation":"",
        "loss":""
    },
    "sniffer":
        {
            "queue":{
                "status":1,
                "delta_t":0.01,
                "intf":"sw2-sw3"
            },
            "traffic":{
                "status":1,
                "intf":"sw2-sw3"
            },
            "socket":{
                "status":0,
                "delta_t":0
            }

        },
    "ip":"10.0.1.2",
    "ftp_server":0,
    "mac":"00:00:00:00:00:02"
},
```


(b) Comutador 2.


```
{
    "type":"SWITCH",
    "name":"sw3",
    "transport_protocol":"bbr",
    "queue":{
        "protocol":"",
        "latency":"",
        "length":"",
        "jitter":"",
        "variation":"",
        "loss":""
    },
    "sniffer":
        {
            "queue":{
                "status":0,
                "delta_t":0.001,
                "intf":"sw3-eth1"
            },
            "traffic":{
                "status":0,
                "intf":"sw3-eth1"
            },
            "socket":{
                "status":0,
                "delta_t":0
            }

        },
    "ip":"10.0.1.3",
    "ftp_server":0,
    "mac":"00:00:00:00:00:03"
},
```


(c) Comutador 3.

Figura 27 – Configuração dos três comutadores utilizados no experimento.

A configuração dos roteadores/comutadores são similares a dos hospedeiros, como visto em seção anterior. Nesse experimento, os comutadores (*switches*) coletam informações do tráfego através de log de *tcpdump* de suas interfaces. O gargalo será forçado pela taxa menor do enlace entre o comutador 2 e o comutador 3. A Figura 27 demonstra como os três comutadores são configurados no framework MINITY.

Considere a configuração do comutador 2 visto na Figura 27b. A indicação de comutador é feita pelo valor *switches* no campo *type*. Observa-se, ainda, que a coleta de dados neste comutador ocorrerá nas interfaces identificadas no campo *sniffer*. A configuração para a extração de informações da fila de saída do comutador serão observadas em *queue* através dos parâmetros *status*, *delta_ t* e *intf*, que representam, repectivamente: um campo booleano para indicar se irá ocorrer a coleta de dados; o tempo de amostragem utilizado para coleta dos dados da interface; e, por fim, o nome da interface de rede em que serão coletados os dados.

Ademais, o tráfego também será coletado, na interface de saída do comutador do gargalo, ou seja, no comutador 2. Todo o tráfego da conexão é então armazenado para futuramente ser utilizado pelas análises. Perceba que o tráfego é coletado em todos os comutadores, entretanto, para afim de análises apenas os dados obtidos no comutador 2 foram utilizados.

Além de especificar os parâmetros dos nós da rede, o arquivo de configuração também é responsável pelo ajuste dos parâmetros das arestas, que são responsáveis pelos atributos dos links entre os nós, como apresentado na Figura 28. São definidos nesse arquivo: os nós que fazem parte desta aresta; o nome de cada interface do nó que irá se comunicar; banda em Mbit/s; a latência acompanhada da unidade de medida; e a taxa de perda de pacote em valor numérico. A Figura 28b descreve a configuração dos enlaces entre os servidores, mostrando a formação do gargalo com a banda de 10 Mbits/s entre os comutadores 2 e 3, enquanto os demais enlaces estão configurados com uma taxa de 20 Mbits/s. Esse gargalo foi criado para testar a adaptação dos protocolos de transporte face a um ponto único de congestionamento compartilhado por todas as aplicações de transferência de arquivos.

```
{
    "type":"EDGE",
    "VERTICE_1":"h1",
    "VERTICE_2":"sw1",
    "intfName1":"h1-sw1",
    "intfName2":"h1-sw1",
    "bw":20,
    "buffer":1500,
    "latency":"0ms",
    "loss":0
},
{
    "type":"EDGE",
    "VERTICE_1":"h2",
    "VERTICE_2":"sw1",
    "intfName1":"h2-sw1",
    "intfName2":"h2-sw1",
    "bw":20,
    "buffer":1500,
    "latency":"0ms",
    "loss":0
},
```

(a) Configuração dos Enlaces Hospedeiros-Comutadores

```
{
    "type":"EDGE",
    "VERTICE_1":"sw1",
    "VERTICE_2":"sw2",
    "intfName1":"sw1-eth4",
    "intfName2":"sw1-sw2",
    "bw":20,
    "buffer":1500,
    "latency":"0ms",
    "loss":0
},
{
    "type":"EDGE",
    "VERTICE_1":"sw2",
    "VERTICE_2":"sw3",
    "intfName1":"sw2-sw3",
    "intfName2":"sw3-eth1",
    "bw":10,
    "buffer":1500,
    "latency":"0ms",
    "loss":0
},
```

(b) Configuração dos Enlaces Comutadores-Comutadores

```
{
    "type":"EDGE",
    "VERTICE_1":"sw3",
    "VERTICE_2":"h3",
    "intfName1":"sw3-h3",
    "intfName2":"h3-eth3",
    "bw":20,
    "buffer":1500,
    "latency":"0ms",
    "loss":0
},
{
    "type":"EDGE",
    "VERTICE_1":"sw3",
    "VERTICE_2":"h4",
    "intfName1":"sw3-h4",
    "intfName2":"h4-eth4",
    "bw":20,
    "buffer":1500,
    "latency":"0ms",
    "loss":0
}
```

(c) Configuração dos Enlaces Comutadores-Hospedeiro

Figura 28 – Configuração dos Enlaces

4.3 ARQUIVO DE PARÂMETRO

O arquivo de parâmetro é responsável pelo controle do experimento, como mencionado em seções anteriores. Neste experimento é simulada a transferência de um arquivo grande entre os hospedeiros clientes e servidores através da rede simulada. Os clientes FTP rodando nos hospedeiros *h3* e *h4* irão solicitar o arquivo nos servidores FTP rodando nos hospedeiros *h1* e *h2*, a partir do instante 1. Eventos de alteração da rede num experimento são efetuados em instantes discretos durante um experimento.

```
[
    {
        "type":"config",
        "n_rodadas":1,
        "tempo_rodada":100
    },
    {
        "time":1,
        "node":"h3",
        "type":"start",
        "ip":"10.0.0.1",
        "filename":"big_file.zip"
    },
    {
        "time":1,
        "node":"h4",
        "type":"start",
        "ip":"10.0.0.2",
        "filename":"big_file.zip"
    },
    {
        "type":"bw",
        "time":20,
        "node":"sw2",
        "intfName":"sw2-sw3",
        "value":20
    },
    {
        "type":"bw",
        "time":40,
        "node":"sw2",
        "intfName":"sw2-sw3",
        "value":10
    },
    {
        "type":"rtt",
        "time":55,
        "node":"h4",
        "intfName":"h4-eth4",
        "value":"120ms"
    },
    {
        "type":"rtt",
        "time":75,
        "node":"h4",
        "intfName":"h4-eth4",
        "value":"40ms"
    }
]
```

Figura 29 – Arquivo de parâmetro em que se configura as ações do experimento

A Figura 29 ilustra a configuração de um experimento, pronto para ser utilizado pelo framework. A configuração mostra que no instante 20 a taxa de transmissão entre os comutadores 2 e 3 será alterada de 10 Mbits/s para 20 Mbits/s, retornando ao valor original de 10 Mbit/s no instante 40. Também será realizada uma alteração no RTT do hospedeiro *h4* que passará de 40 ms para 120 ms e retornará ao valor original no instante 75. O tempo total do experimento será ajustado para 100 segundos, encerrando o experimento passados 100 segundos do seu início. Há que observar que o experimento acontece a partir de um estado inicial em que todas as filas nas interfaces estão completamente vazias, evoluindo para outros estados com o decorrer do experimento. Se for interesante ter uma ideia do comportamento em equilíbrio de um determinado cenário de rede, apenas é necessário adotar um tempo de experimento adequado.

Esta configuração exemplo serve para demonstrar a versatilidade do framework MINITY na manipulação de parâmetros para a criação de um experimento complexo. Além disso os eventos executados por este experimento, aumento e diminuição de RTT e variação da banda disponível no gargalo, tem como objetivo permitir analisar a reação de protocolos alvo a constantes alterações do tráfego na rede, como ocorre em um cenário real.

4.4 ANÁLISE DE RESULTADOS

Dado que o objetivo do framework MINITY é fornecer um ambiente que seja totalmente automatizado, desde a sua configuração até a exploração dos dados, após o usuário definir os parâmetros e configurar o ambiente para a execução, será possível realizar a análise dos dados através de gráficos. Serão investigados os dados seguintes: taxa de envio, fila de pacotes e comportamento das variáveis de controle do protocolo de transporte BBR, usado como protocolo alvo para as transferências de arquivo no cenário do experimento.

4.4.1 Taxa de Envio e Fila de Pacotes

A taxa de envio é medida através de amostragem em que se contabiliza a quantidade de bytes que está sendo transmitida por segundo. A qualquer momento, o protocolo TCP BBR pode estar medindo erroneamente a taxa de banda disponível no enlace, que será estimada em função da perda de pacotes ou de congestionamento das filas ao longo da infraestrutura da rede.

Para acompanhar a taxa de envio real de cada hospedeiro servidor, utilizamos a função *plotSendingRate* para gerar o gráfico da taxa de envio do nó.

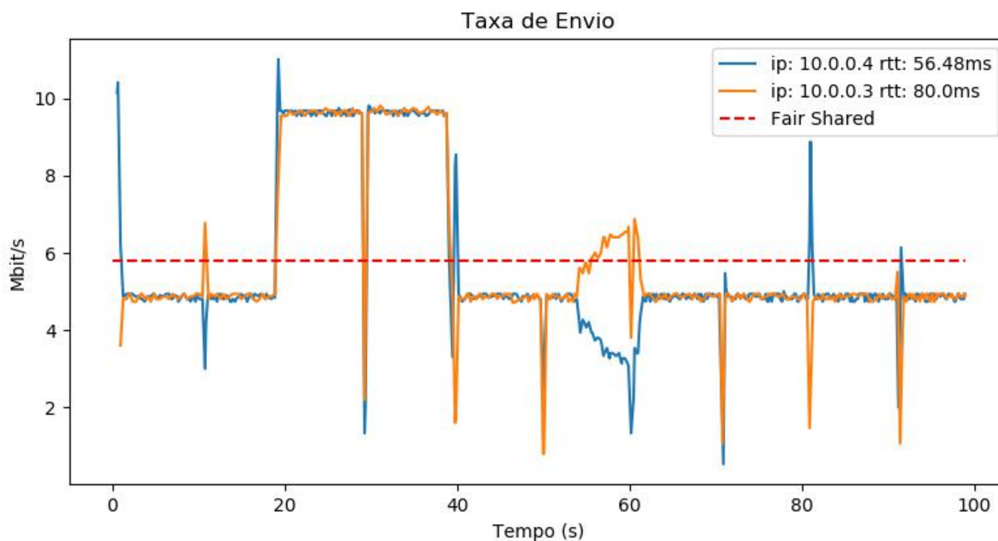


Figura 30 – Taxa de envio entre dois fluxos TCP BBR. Essa taxa é capturada a partir da informação dos pacotes TCP que passam pelo gargalo da rede. É realizada uma estimativa média de quantos bits/segundo estão sendo transmitidos a partir do tamanho dos pacotes TCP.

O gráfico do exemplo utilizado por este trabalho pode ser visto na Figura 30, em que dois fluxos TCP BBR compartilham um enlace, inicialmente, com 10 Mbit/s e, no instante 20, a capacidade de envio do enlace é alterada para 20 Mbit/s, retornando aos 10 Mbps/s no instante 40. A linha tracejada em vermelho indica o *fair shared*, ou seja, metade da média da capacidade do enlace ao longo do experimento. Pode-se observar na figura que as duas conexões apresentam comportamento médio semelhante, convergindo apesar dos comportamentos adaptativos opostos entre 50 e 60 segundos. Os picos nas taxas em intervalos de mais ou menos 10s são devidos aos parâmetos utilizados pelo protocolo BBR para tentar operar em uma taxa de banda superior. Os rótulos neste gráfico indicam também o IP do Hospedeiro cliente e a média do *Round Trip Time* durante o experimento.

Já na Figura 31 é possível observar o gráfico da fila onde está o gargalo da conexão. Esse dados - da fila do gargalo - foram obtidos através de uma amostragem coletada em uma quantidade de tempo definida pelo usuário no arquivo de configuração. Neste experimento foi utilizado o valor de 0.01 segundos. Esse gráfico pode ser interessante para análisar se as filas dos equipamentos estão sobrecarregadas e se seus tamanhos estão definidos erroneamente. Consequentemente, essa sobrecarga nas filas pode causar uma variação na latência, reduzindo a taxa de transferência global da rede e aumentando a latência. Esse problema é conhecido como *bufferbloat* (GETTYS, 2011). É possível utilizar esse gráfico, do tamanho do buffer, na Figura 31, e o gráfico da taxa de envio, Figura 30, para entender como o protocolo reage às alterações da rede. Neste trabalho foi considerado apenas as filas das interfaces de saída do gargalo da conexão, o comutador 2. Todavia pode-se estender esta análise para todos os equipamentos da infraestrutura que estão sendo emulados.

A fim de compreender a diferença entre como os protocolos reagem ao preenchimento

da fila do gargalo da conexão, executou-se um mesmo experimento para comparar a sua utilização no caso do TCP BBR com os protocolos TCP CUBIC e TCP NewReno. Pode-se observar, através das Figuras 31, 32 e 33, que o TCP BBR obteve êxito em manter o preenchimento da fila estável e com uma pressão menor quando comparado aos outros protocolos. Considerando que o tamanho de pacote seja de 1500 bytes, o TCP BBR manteve na fila do gargalo em torno de 8 pacotes durante toda a conexão. Já o TCP NewReno manteve 10 pacotes e reduziu para em torno de 8 pacotes, durante o aumento do RTT de 40 ms para 120 ms, enquanto durante a redução do RTT no instante 75 a quantidade de pacotes enfileirados se tornou estável em torno de 12 pacotes, um aumento quando se comparado ao comportamento inicial. Por fim, o TCP CUBIC manteve a fila do gargalo com 10 pacotes e durante a alteração do RTT ocorreu uma redução na quantidade de pacotes enfileirados, para 8 pacotes, e permanece até o fim do experimento.

Vale ressaltar que o controle de tráfego realizado pelo framework MINITY consiste de dois algoritmos implementados em cada interface dos equipamentos da rede, como apresentado em seção anterior. Uma fila para o controle da banda e outra fila para a adição de latência, taxa de perda de pacotes e enfileiramento dos pacotes. A amostragem obtida pelo framework MINITY é da fila 2 enquanto a fila 1 limita a quantidade de tráfego. A fila de controle de banda têm como parâmetro a quantidade máxima de bytes que poderá ser recebido por uma rajada sem a ocorrência de descarte. Com esta parametrização, caso o protocolo esteja enviando uma quantidade de banda superior na interface que possui o controle de banda, pacotes serão descartados na $qdisc_1$.

Sabe-se que o TCP NewReno e o TCP CUBIC têm o seu controle de congestionamento baseado no descarte de pacotes. Portanto, os resultados demonstrados nas Figuras 32 e 31 indicam descartes de pacotes antes do preenchimento da segunda fila, comportamento esperado, já que ambos os protocolos são sucetíveis ao problema de bufferbloat (GETTYS, 2011). Já o TCP BBR manteve uma pressão menor na fila existente no gargalo quando comparado aos outros prtocolos.

Para estender a análise do comportamento dos protocolos frente ao errôneo dimensionamento das filas do gargalo da conexão, adicionou-se um experimento complementar no apêndice A.

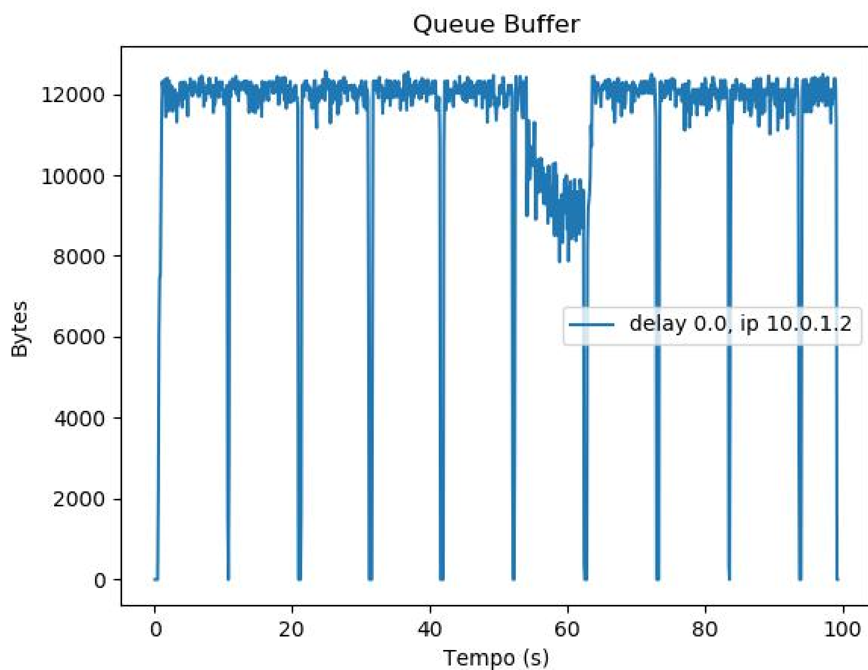


Figura 31 – Fila de pacotes do gargalo, na interface de saída do comutador 2. Duas filas estão presentes no gargalo da conexão, uma para o controle da taxa de envio de pacotes e outra para a adição de latência, taxa de perda e enfileiramento dos pacotes. Percebe-se que apenas 12000 bytes foram utilizados pelo protocolo TCP BBR. Isso se deve a descartes estarem ocorrendo na primeira fila, a de controle de banda. Mesmo assim, pode-se observar o TCP BBR utilizou apenas 12000 bytes, sempre constantes. Esse resultado se difere de outros protocolos, que dão uma sobrecarga maior na fila. Por fim, durante o acréscimo do RTT, houve uma redução da quantidade de pacotes enfileirados.

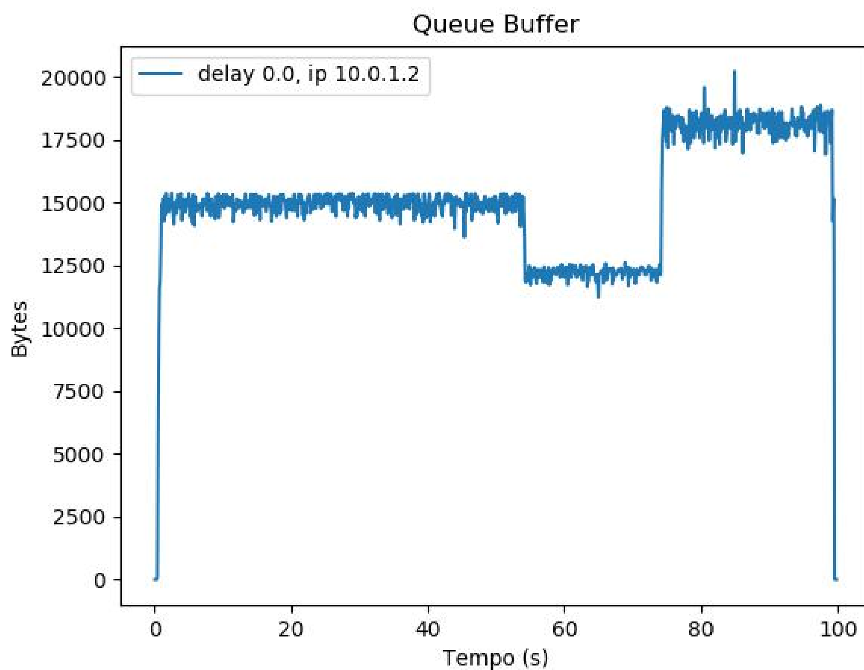


Figura 32 – Fila de pacotes do gargalo, na interface de saída do comutador 2. Duas filas estão presentes no gargalo da conexão, uma para o controle da taxa de envio de pacotes e outra para a adição de latência, taxa de perda e enfileiramento dos pacotes. Pode-se perceber que 15000 bytes foram utilizados pelo protocolo TCP NewReno, valor limite para a quantidade de rajadas aceitas inicialmente pelo controle de tráfego de MINITY. Ademais, após aumento do RTT, mais pacotes foram armazenados na segunda fila do gargalo da conexão.

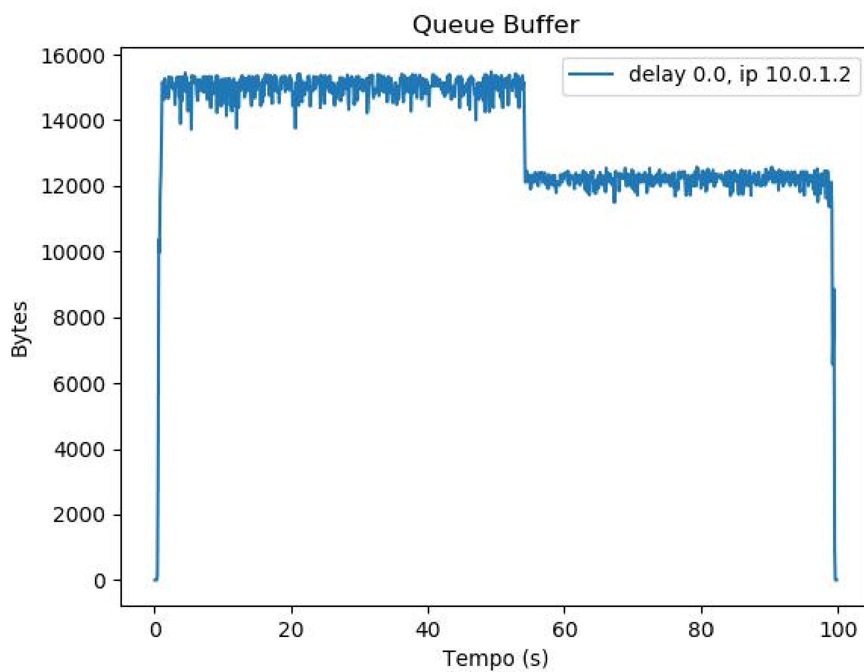


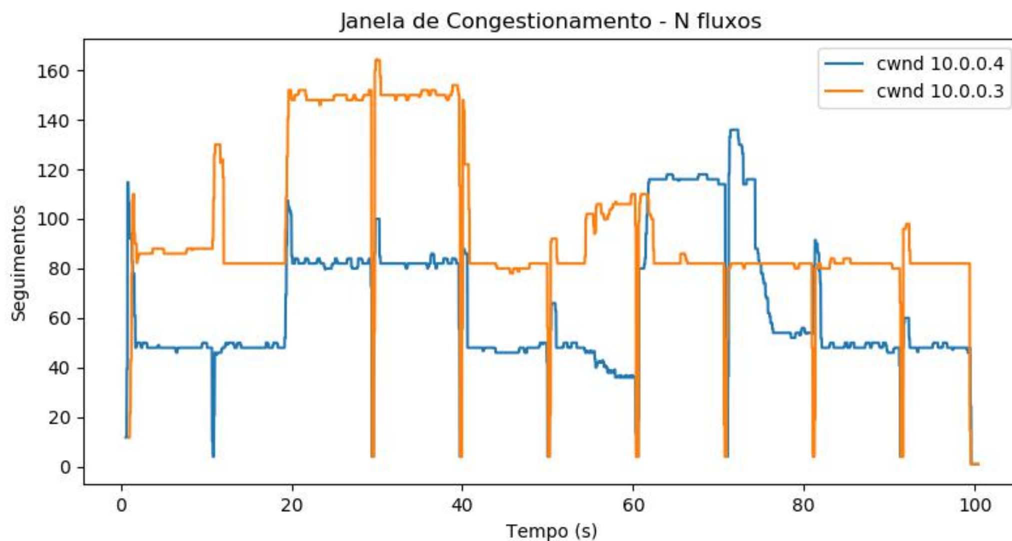
Figura 33 – Fila de pacotes do gargalo, na interface de saída do comutador 2. Percebe-se que o TCP Cubic manteve a fila em torno de 15000 bytes.

4.4.2 Variáveis de Controle do Protocolo

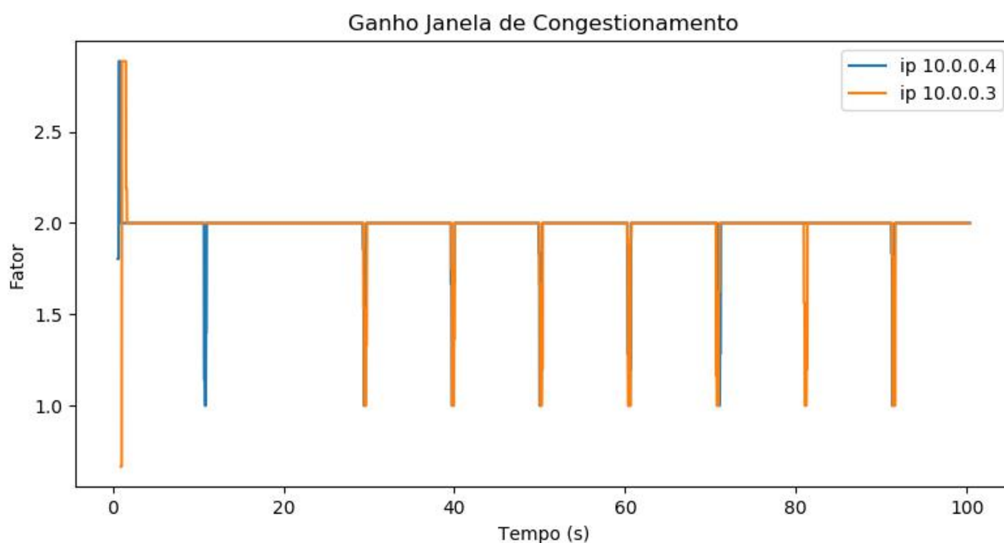
Devido ao interesse deste trabalho ser focado na análise do TCP BBR, é automatizada a geração de gráficos de algumas de suas variáveis internas. Na seção 2 foi definido e explicado o significado de cada variável de controle deste protocolo.

As Figuras 34 e 35 contém os gráficos de todas as métricas necessárias para avaliar o protocolo TCP BBR. São informações obtidas, em tempo de real, das variáveis de controle do protocolo de dados reais.

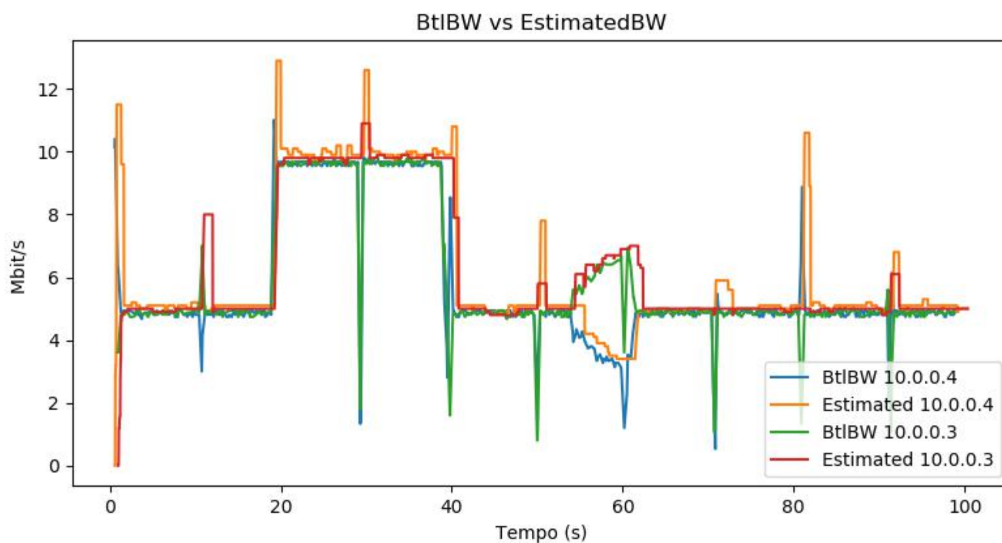
O protocolo TCP dispõe de algumas variáveis para o controle de congestionamento, como a janela de congestionamento, que é mantida no servidor em que se está enviando dados. Assim, caso o cliente esteja com a sua conexão sobrecarregada, é diminuída a quantidade de segmentos a serem enviados. Esta variável pode ser observada através do gráfico da *Janela de Congestionamento*, exemplificado na Figura 34a. Já na Figura 34b indica a alteração da variável *Congestion Window Gain* que controla a taxa de aumento da janela de congestionamento do TCP BBR, pode-se observar que a cada 10 ciclos há uma redução para 1, o que indica que o TCP BBR entrou na fase de *Drain*. Na Figura 34c se encontra o gráfico com as informações, dos dois fluxos utilizados no experimento, da taxa medida (EstimatedBW) pelo protocolo BBR e a taxa real de envio (BtlBW) que passa ao longo do gargalo da conexão. Já na Figura 35a observa-se o RTT e o MRTT sendo respectivamente, o Round Trip Time e a variável utilizada pelo TCP para estimar o RTT. Por fim, na Figura 35b se encontra a variável *Pacing Gain*, é através dela que o protocolo determina a taxa de envio do protocolo, pode-se observar que a cada 10 ciclos o seu valor é reduzido para o inverso do valor utilizado, ou seja, se for de 1.25 (ganho de 25%) o valor durante a redução será de 0.75 (redução de 25%).



(a) Janela de Congestionamento

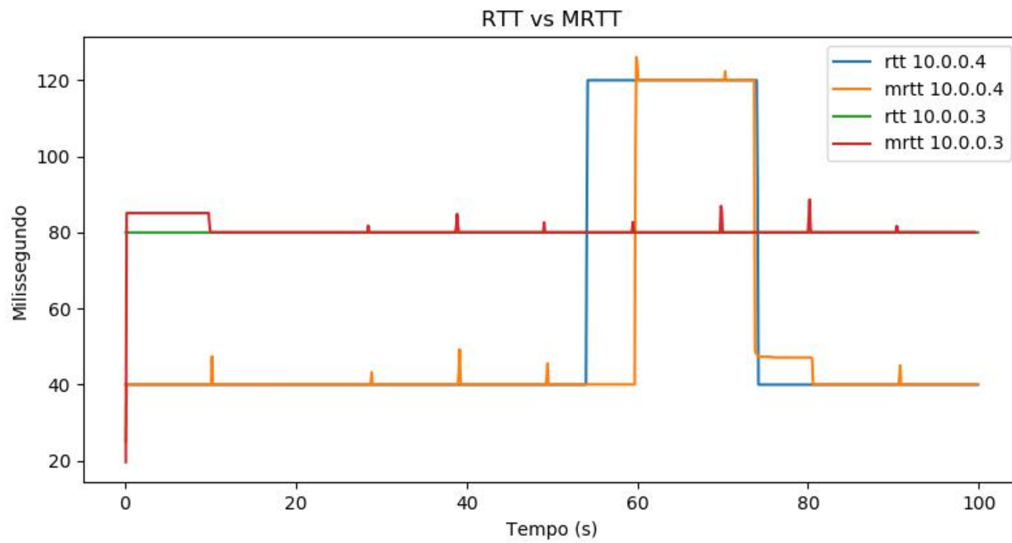


(b) Ganho da Janela de Congestionamento

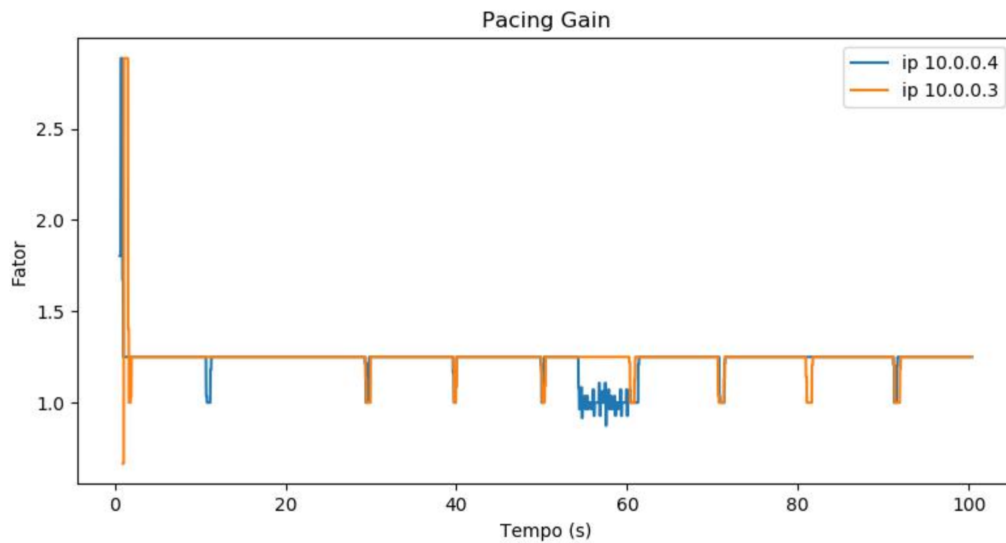


(c) Banda Estimada pelo Protocolo

Figura 34 – Variáveis de Controle do protocolo BBR. Em (a) encontra o gráfico da janela de congestionamento do TCP, dado pela quantidade de segmentos ao longo do experimento. Em (b) demonstra a variação da variável do Ganho da Janela de Congestionamento. Já em (c) pode-se observar o gráfico da banda estimada pelo BBR vs a banda utilizada pelos fluxos.



(a) Medição do *Round Trip Time*



(b) Variável Pacing Gain

Figura 35 – Variáveis de Controle do protocolo BBR. Em (a) encontra-se a variável de medição do RTT vs o RTT real dos fluxos. Em (b) a variável pacing gain dos fluxos.

O protocolo TCP BBR também disponibiliza algumas variáveis para o controle de congestionamento. Estas variáveis armazenam medições que o protocolo efetua durante a transferência de dados. Através destas medições, avalia-se a disponibilidade de banda e RTT para que, assim, possa ser efetuada uma decisão de aumento ou de diminuição da taxa de envio de pacotes.

5 MINITY EM AÇÃO: PERFORMANCE COMPARATIVA TCP BBR

Neste capítulo serão comparados os resultados do ambiente MINITY com os resultados do framework inspirador deste trabalho (JAEGER et al., 2019) e checar se MINITY reproduz os mesmos resultados dos experimentos do trabalho seminal. Na seção 1 será detalhado um experimento a partir de um único fluxo do TCP BBR. Na seção 2 serão averiguados os resultados de ambos os frameworks a partir de múltiplos fluxos BBR, dividos em dois grupos com RTTs diferentes.

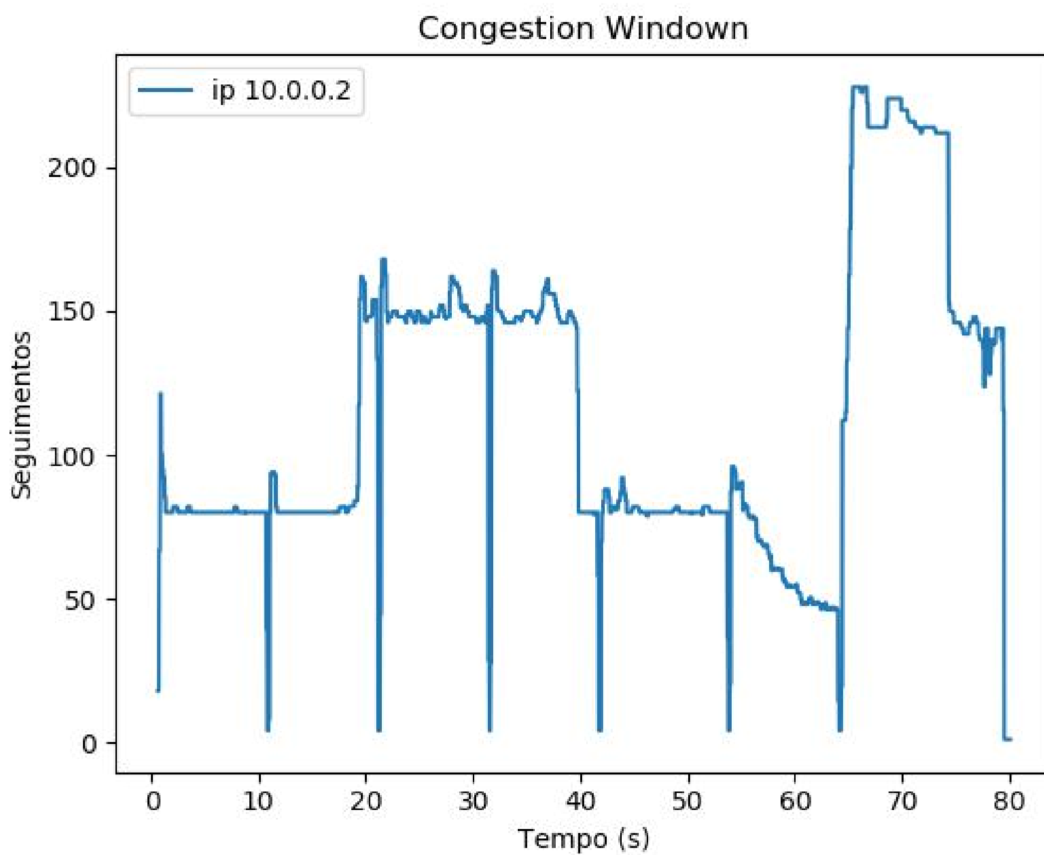
5.1 UM ÚNICO FLUXO BBR

Nesse experimento queremos sondar o tempo de reação do TCP BBR em relação a mudanças na rede. Coletamos algumas métricas inerentes ao envio e da condição da rede em todo o experimento. Realizamos algumas alterações nas condições da rede e coletamos os valores das variáveis de controle do BBR. Ao final do experimento comparamos com os resultados obtidos no framework (JAEGER et al., 2019).

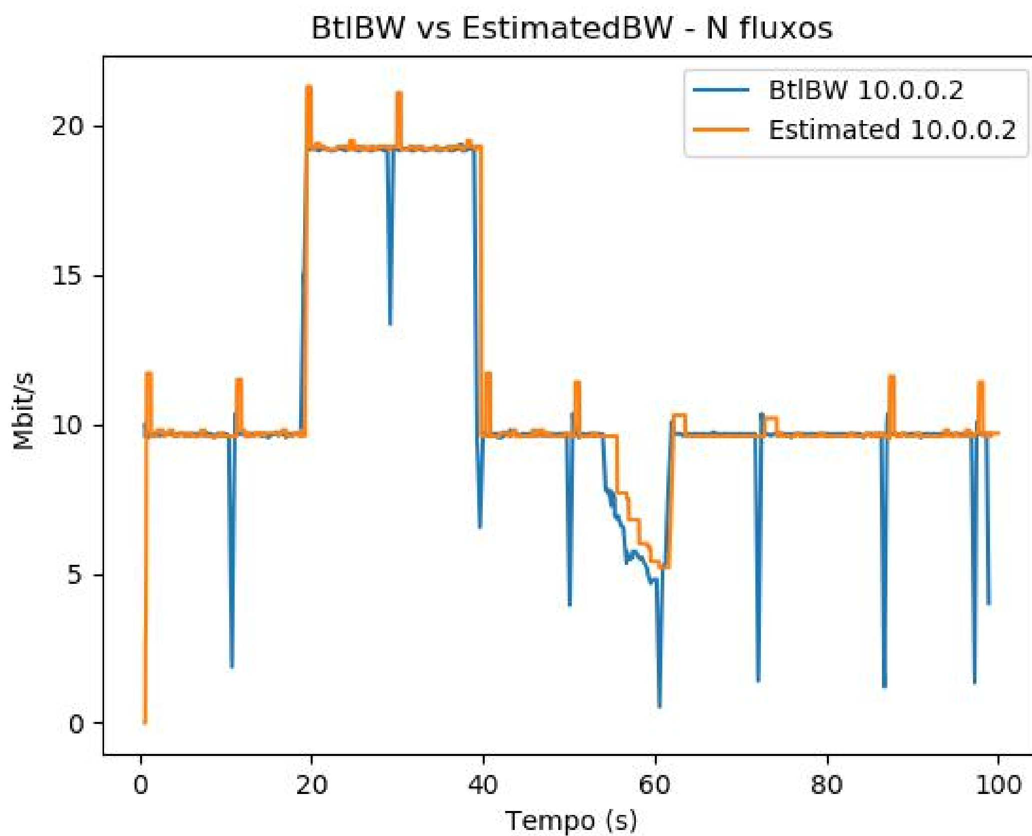
O tempo total do experimento foi de 80 segundos, iniciando com uma banda de 10Mbit/s, RTT de 40 ms e com o tamanho de buffer ajustado para 2*BDP (34 pacotes de 1500 bytes). No instante de 20 segundos após início da conexão, a quantidade de banda disponível no gargalo foi alterada de 10 Mbit/s para 20 Mbit/s e no instante 40 segundos esse valor foi retornado para o valor inicial. Esta alteração de banda ocorre frequentemente na internet, por inúmeros motivos, como, por exemplo, mudança de rota no encaminhamento de pacotes ou pelo aumento de fluxos que estão compartilhando o gargalo. Logo, realizar experimentos que simulem esse comportamento é fundamental para que se possa compreender o tempo de reação do protocolo BBR em estimar a banda disponível do gargalo da conexão e o RTT.

Na Figura 36b, a linha azul mostra a taxa de envio do hospedeiro servidor para o hospedeiro cliente, enquanto a linha laranja mostra o valor de banda estimado pelo BBR. Percebe-se que o BBR obteve êxito em estimar a banda no momento em que a rede sofre alterações, tanto no acréscimo de 10 Mbit/s para 20 Mbit/s no instante 20 segundos, quando do decréscimo de 20 Mbit/s para 10 Mbit/s no instante de 40 segundos. No entanto, podemos perceber um comportamento anômalo no instante de 55 segundos, em que a variável de medição do BBR e a taxa de envio do hospedeiro servidor são reduzidas consideravelmente. Este comportamento do BBR é causado pela alteração do RTT. Como a medição da taxa de envio é realizada a todo instante pelo protocolo, o atraso no recebimento da confirmação dos pacotes provoca a redução da banda estimada, reduzindo a taxa de envio. Entretanto, o valor da medição de banda retorna à normalidade após o protocolo BBR entrar na fase de *Probe RTT* e efetuar a estimativa correta do RTT.

Na Figura 37a podemos observar o *round trip time* da rede na linha azul e o valor medido pelo protocolo *MRTT* na linha laranja. Ao longo do experimento ocorre uma alteração do RTT no instante de 55 segundos. Entretanto, pode-se observar que o *MRTT* teve atraso de cerca de 5 segundos para reconhecer esta alteração. Isso se deve ao fato de que a medição do RTT é realizada apenas a cada 10 segundos, momento no qual o protocolo BBR sai da fase *Probe Bandwidth* e entra na fase de *Probe RTT*, devido a validade do RTT estimado se esgotar. A alteração do RTT gerou impacto também na estimativa de banda, como pode ser visto na Figura 36b. Por volta do instante 55 segundos, a taxa de envio cai consideravelmente, tendendo a zero em certo instante de tempo, sendo um indicativo de que o BBR entrou na fase de *Drain*, que possibilita esvaziar uma fila em possível gargalo. Após a medição do RTT, no instante 60, o BBR então refaz sua estimativa e o valor de transmissão retorna à normalidade.

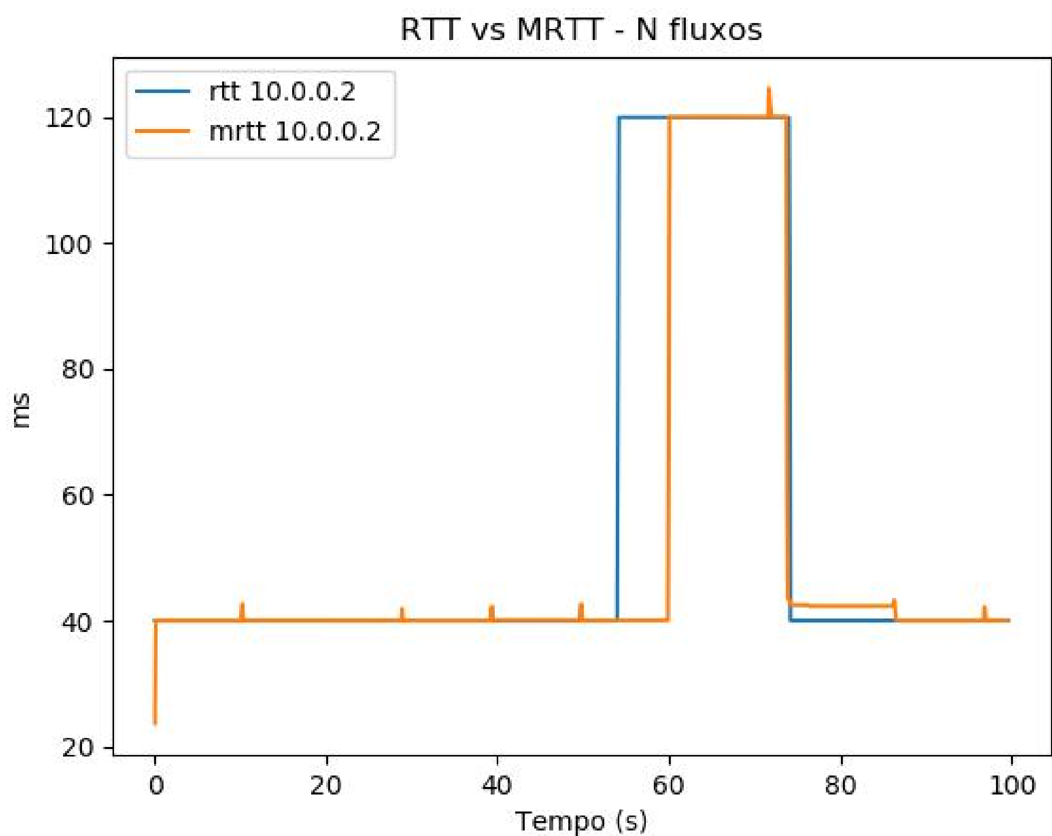


(a) Janela de Congestionamento

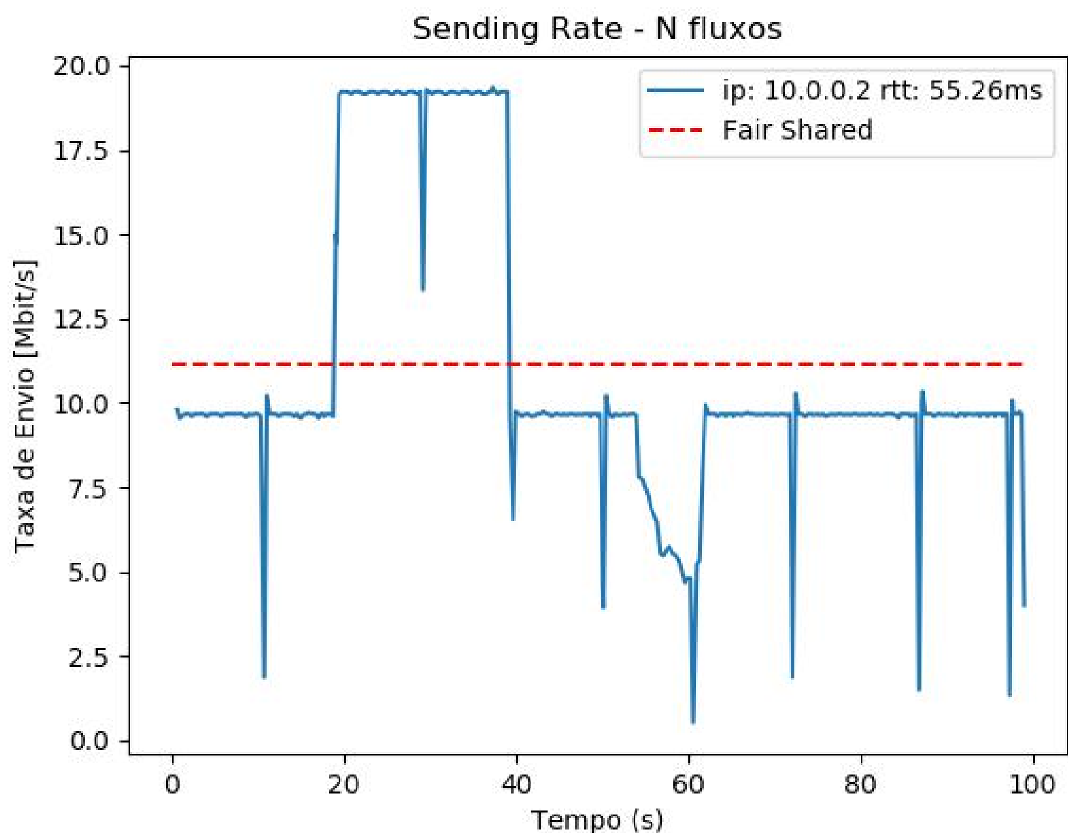


(b) Banda Estimada pelo Protocolo

Figura 36 – Variáveis de Controle e Medições realizadas através do framework MINITY. Em (a) o resultado da janela de congestionamento. Em (b) o resultado da banda do gargalo da conexão vs valor de banda estimado pelo protocolo BBR.



(a) Medição do Round Time Trip



(b) Taxa de envio

Figura 37 – Variáveis de Controle e Medições realizadas através do framework MINITY. Em (a) encontra-se o resultado da medição do RTT (MRTT) vs RTT real da rede. Em (b) a taxa de envio calculado a partir da quantidade média de pacotes em um determinado intervalo de tempo.

5.1.1 Comparação entre os resultados

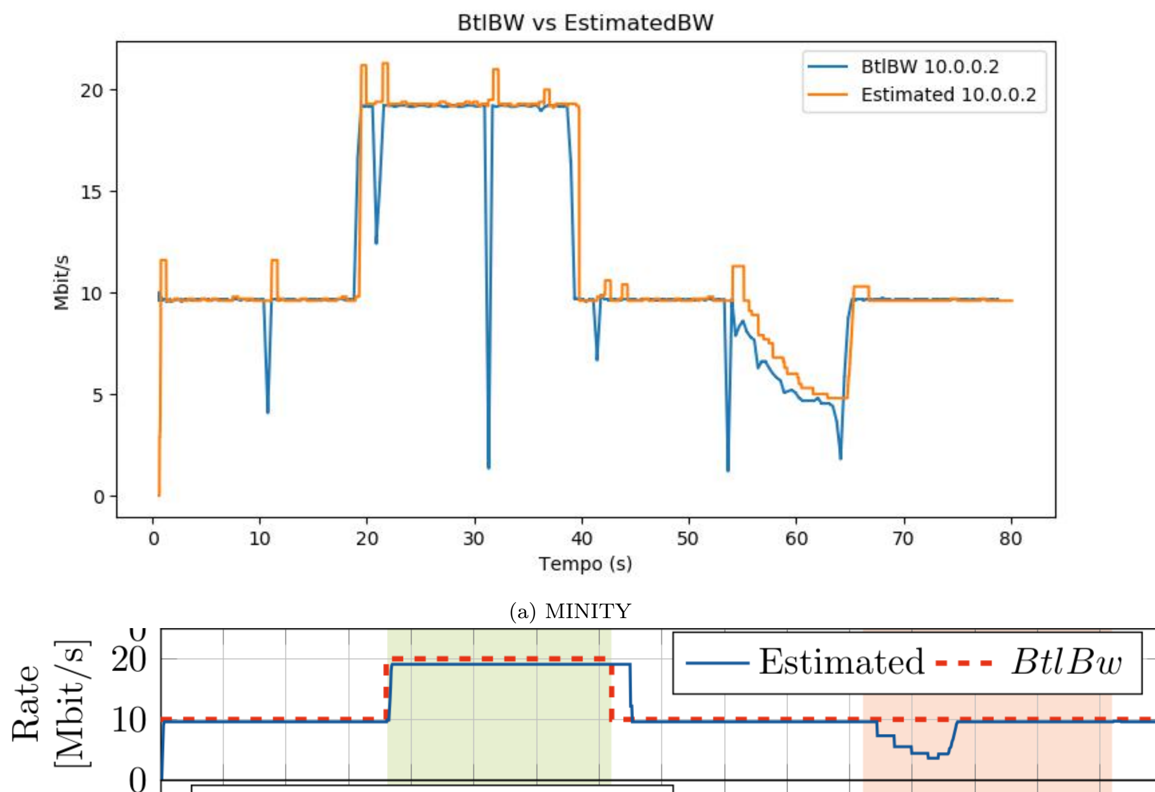
As Figuras 38a e 38b mostram a taxa de envio da conexão medida pelo MINITY e por (JAEGER et al., 2019), respectivamente. Nas figuras são mostradas a estimativa da banda disponível pelo protocolo BBR e a variável BtlBW. A estimativa da banda média é obtida amostrando o gargalo da conexão.

PHAR: Não existe a abcissa da figura do Jaeger e deveria estar ali para referenciar, já que isso correlaciona as figuras (a) e (b).

No framework MINITY plotou-se a taxa real de transmissão na linha azul e a variável BtlBw na cor laranja. Já em (JAEGER et al., 2019) foi plotado o valor máximo da banda entre os hospedeiros na linha vermelha e a variável BtlBw em azul. Podemos observar comportamentos semelhantes nos momentos em que ocorre o aumento ou a diminuição da taxa de envio nos instantes 20 e 40, quando o BBR reage com ligeiro atraso na estimativa da banda real.

Ademais, observa-se que picos na linha laranja ocorrem durante a transferência, isso se deve ao fato do BBR entrar na fase de *Probe Bandwidth*, fazendo com que, a cada início desta fase, o TCP BBR ajuste o *pacing gain* para o valor de 1.25, seguido por uma drenagem na fila com a proporção inversa. Já os picos capturados na linha azul, que captura a taxa de envio no gargalo, é consequência do protocolo entrar na fase de *Probe RTT* em que o TCP BBR reduz a taxa de envio para 4 pacotes com a finalidade de drenar possíveis filas criadas para capturar a estimativa real do RTT. A captura deste comportamento, demonstra que MINITY logrou êxito em capturar fases do TCP BBR - diferente do framework proposto por (JAEGER et al., 2019) - isso se deve à flexibilidade do usuário configurar a quantidade de tempo para se realizar amostragem, seja da variável estimada do protocolo BBR quanto da estimativa de banda real que trafega pelo gargalo. Observa-se que, através da manipulação desse tempo de amostragem, pode-se obter o comportamento dos ciclos do TCP BBR com mais fidedignidade.

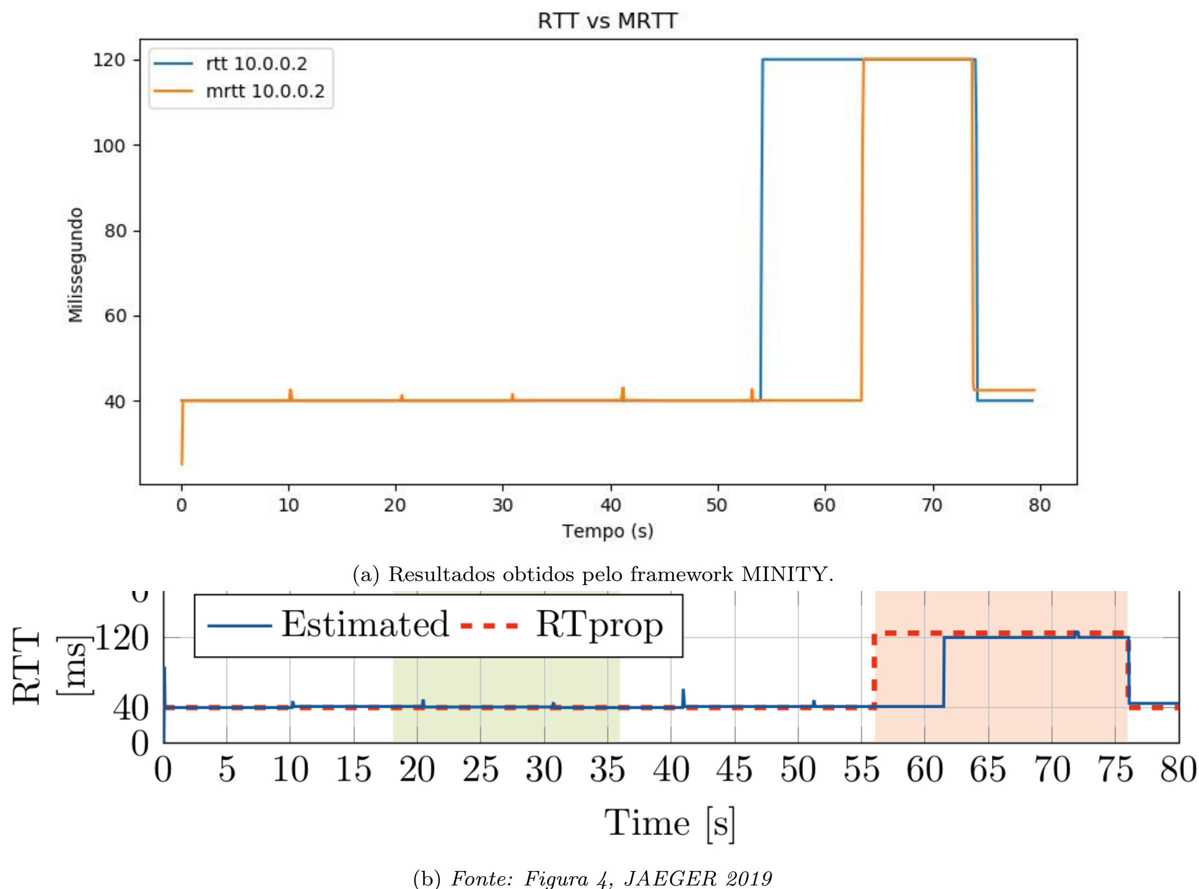
Além disso, o comportamento no instante 55 segundos, quando o RTT é alterado de 40 ms para 120 ms, é também semelhante. O TCP BBR, acreditando que o aumento de RTT tenha sido causado por um tráfego maior no enlace com aumento do congestionamento, reage reduzindo a sua estimativa da banda disponível, até que reavalie que a banda disponível não se encontra alterada. Este é o comportamento adaptativo do TCP BBR tentando estimar a taxa de envio a ser usada. Fica claro que a medição de banda do BBR é prejudicada por uma alteração repentina do RTT entre os hospedeiros.



(a) MINITY

(b) Banda estimada pelo BBR vs banda disponível no gargalo da rede,*Fonte:(JAEGER et al., 2019), Figura 4.*

Figura 38 – Comparação entre os resultados da taxa de envio do framework MINITY com o framework desenvolvido em (JAEGER et al., 2019).



(a) Resultados obtidos pelo framework MINITY.

(b) *Fonte: Figura 4, JAEGER 2019*

Figura 39 – Comparação do resultado da alteração do RTT entre os frameworks MINITY e (JAEGER et al., 2019).

Além da variável de medição da banda, pode-se utilizar a medição do RTT e da variável *MRTT*, ou de *RTprop*, como referenciado em (JAEGER et al., 2019) .

No próximo experimento comparativo o RTT inicial é de 40 ms. No instante 55 segundos é incrementado para 120 ms e no instante 75 é reduzido novamente para o valor inicial de 40ms. Podemos observar que ambos os frameworks obtiveram os mesmos resultados, como pode se observar na Figura 39, a alteração do RTT demorando alguns segundos até ser detetada, como mostra a linha laranja na Figura 39a e a linha azul na Figura 39b. No instante 75, em que o valor do RTT retorna para o valor inicial de 40 ms, o BBR realiza esta medição eficientemente, sem atraso, e ambos frameworks obtiveram os mesmos resultados.

Este comportamentos semelhantes dão a entender que as versões de TCP BBR implementadas são as mesmas ou de comportamento muito similar.

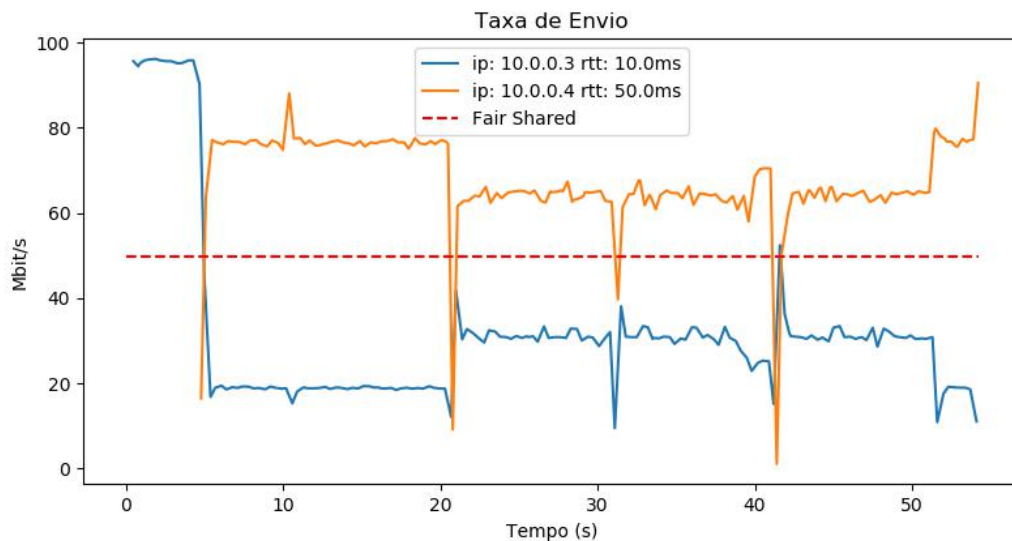
5.2 MÚLTIPLOS FLUXOS BBR

Nesta seção iremos averiguar o comportamento de vários fluxos BBR, com diferentes RTTs, compartilhando o mesmo gargalo de enlace. O objetivo é verificar se há alguma

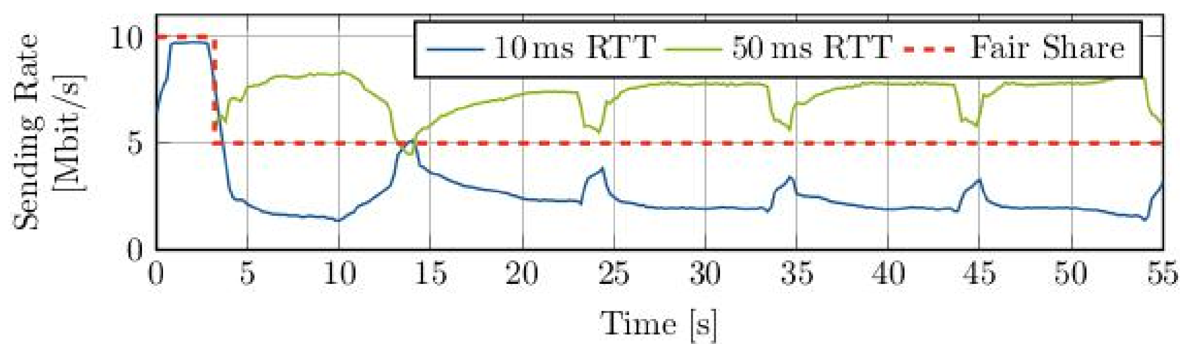
distinção no compartilhamento de banda entre os fluxos e a sua proporcionalidade com o RTT.

5.2.1 2 Fluxos BBR

Neste experimento utilizou-se dois fluxos com 10 ms e 50 ms de RTT, respectivamente, compartilhando um enlace com 100 Mbit/s. O fluxo com RTT maior será inicializado após 5 segundos do início do experimento. Em (JAEGER et al., 2019) foi realizado o mesmo experimento, no entanto, com uma banda de 10 Mbits. Na Figura 40b, do artigo, mostra que o fluxo com RTT de 10 ms apresenta uma taxa de envio em torno de 2,0 Mbps, enquanto o fluxo com RTT de 50 ms consegue algo em torno de 7,5 Mbps, quase 4 vezes mais. No nosso cenário com taxa de 100 Mbits/s, os resultados obtidos pelo MINITY e mostrados na Figura 40a indicam que o fluxo com RTT de 10 ms se mantém com uma taxa de banda em torno de 20 Mbit/s, enquanto o fluxo com RTT de 50ms, obtém em torno de 80Mbit/s ao longo dos 20 primeiros segundos do experimento, um fator de 4x mais em comparação com o fluxo de RTT menor, em conformidade com os resultados anteiores. Após os primeiros 20 segundos, o BBR realiza aferições de banda e com isso há uma diminuição da diferença de utilização da banda entre os fluxos. O fluxo de 10 ms passa operar em torno de 35 Mbit/s enquanto o fluxo de 50ms em torno de 65 Mbit/s, agora com um fator em torno de 2x mais apenas. Esse comportamento do BBR a favor de fluxos com RTTs maiores é oposto ao controle clássico do TCP baseado em perda que aloca uma maior banda no gargalo para fluxos com RTTs menores (BROWN, 2000). Este resultado demonstra que novos experimentos precisam ser realizados para permitir uma análise mais profunda da correlação dos parâmetros do BBR en cenários com múltiplos fluxos, variados RTTs e diferentes capacidades dos enlaces. Para isso, MINITY poderá ser de extrema ajuda pela versatilidade e facilidade de obtenção de resultados com uso de implementações reais de BBR.



(a) Resultado obtido pela utilização do framework MINITY.



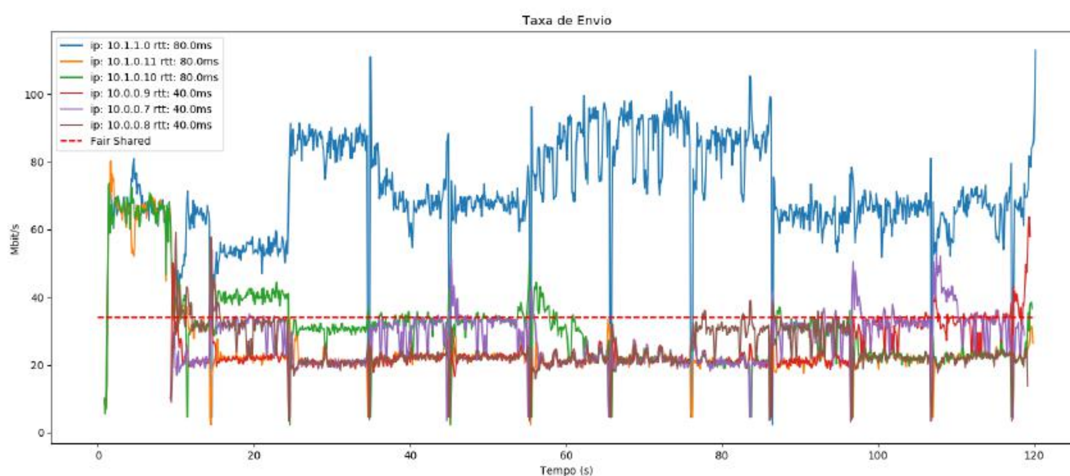
(b) *Fonte: Figura 5a, JAEGER 2019*

Figura 40 – Comparação dos resultados entre os frameworks. O objetivo é identificar o comportamento do protocolo BBR em um cenário com fluxos com diferentes RTTs.

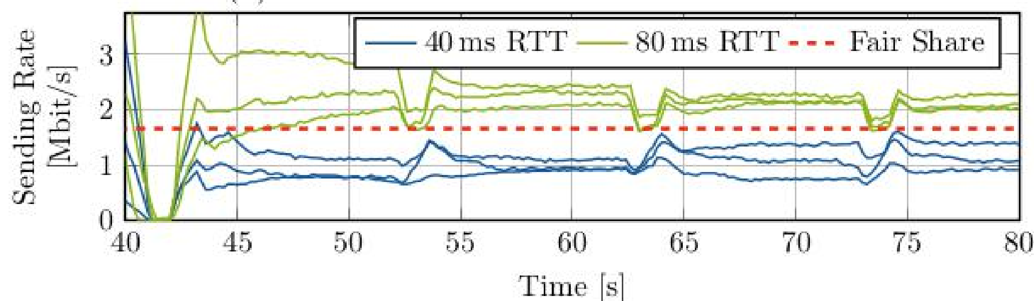
5.2.2 6 Fluxos BBR

Nesse experimento reportado em (JAEGER et al., 2019), consiste de 6 fluxos BBR, dividos em dois grupos usando RTT de 40 ms e de 80 ms. Alguns parâmetros não foram descritos pelos autores, de forma que algumas características utilizadas não foram as mesmas.

Para emular no ambiente MINITY, utilizou-se 200 Mbits/s de banda, e os fluxos com 40 ms se iniciam após 10 segundos do início dos fluxos de 80 ms.



(a) Resultado obtido com a utilização do framework MINITY. 6 Fluxos são divididos em dois grupos de RTT com 40 ms e 80 ms respectivamente, compartilhando um gargalo com 200 Mbit/s.



(b) *Fonte: Figura 5b, JAEGER 2019*

Figura 41 – Comparação dos resultados entre os frameworks MINITY e o desenvolvido por (JAEGER et al., 2019) de um experimento com 6 fluxos TCP BBR divididos em dois grupos de RTT distintos com 40 ms e 80 ms.

Como foi evidenciado em (JAEGER et al., 2019), os grupos de mesmo RTT tendem a se manter com taxas de envio semelantes, como mostra o resultado na Figura 41b. Entretanto, essa observação foi reportada apenas por um intervalo limitado de tempo, entre os instantes 40 e 80. Além disso, é reportado que fluxos com maiores RTTs tendem a dispor de mais banda do que fluxos com RTT menores.

Ao simular esse experimento no ambiente MINITY, como pode ser encontrado na Figura 41a, o resultado encontrado foi diferente do ponto de vista da sincronização dos

fluxos.

Pode-se observar uma sincronização entre os fluxos com baixo RTT em relação a disputa pela taxa de banda do gargalo. Já os fluxos com maior RTT são responsáveis pela aquisição de uma maior quantidade de banda. Além disso os fluxos de 80 ms não se sincronizam e apenas um dos fluxos, o de linha azul, consome quase que 50% da banda disponível do gargalo. Enquanto o fluxo verde, também de 80 ms, consome ora 40 Mbit/s ora 20 Mbit/s, dependendo do momento da observação. Já o fluxo laranja, também de 80 ms, sincroniza com os fluxos de baixo RTT e consome 20 Mbit/s em toda a sua transmissão. Já os fluxos com RTT de 40ms, representados pelas linhas vermelhas, roxas e marrom, se sincronizam em quase todo o experimento ao utilizarem cerca de 20 Mbit/s, valor inferior ao compartilhamento justo (Fair Shared) de banda, representado pela linha tracejada, de aproximadamente de 33 Mbit/s.

Alguns fatores podem causar a diferença encontrada nos resultados. Como foi reportado, o experimento conduzido em (JAEGER et al., 2019) limtou-se a analisar apenas 20 segundos de todo o experimento, enquanto nosso trabalho reportou os dados obtidos desde o início do experimento até o seu fim no instante 120 segundos. Além disso, variáveis como tamanho dos buffers também não foram reportadas e optamos por utilizar o valor de 2*BPD. Portanto é necessário uma investigação mais extensa para averiguar se o comportamento de sincronização dos fluxos, como reportado em (JAEGER et al., 2019), de fato ocorre em quaisquer circunstâncias. E mais uma vez MINITY poderá ser de grande valia.

6 CONCLUSÃO E TRABALHOS FUTUROS

Neste trabalho de conclusão de curso desenvolveu-se o framework MINITY, que tem como objetivo a emulação de um ambiente de rede real apropriado para a experimentação de variações de controle de congestionamento para o protocolo TCP.

No capítulo 2 revisa-se, sistematicamente, as diferentes soluções para o controle de congestionamento do TCP, com o intuito de demonstrar a diversidade para a solução do problema de congestionamento. Foram abordados os TCPs clássicos, Reno e NewReno, o TCP CUBIC e, por fim, TCP BBR, o protocolo motivador deste trabalho.

O capítulo 3 descreve o framework MINITY, levando em consideração práticas de desenvolvimento de software, como programação orientada a objetos, modularização e a simulação de um ambiente real. Essas características permitem que futuros desenvolvedores possam adicionar funcionalidades ao framework sem a necessidade de grandes alterações no código. Além disso, MINITY fornece ao usuário uma interface simples para o ajuste de configuração do experimento. Tais características se sobressaem quando comparado a outros frameworks ou a ferramentas de simulação como a *ns-3* (RILEY; HENDERSON, 2010).

O capítulo 4 detalha a ambientação de um experimento minuciosamente, a fim de auxiliar os futuros usuários. Desde a concepção dos arquivos de configurações do ambiente e do experimento, até a geração dos gráficos para as análises. Neste capítulo também é possível observar a facilidade ao manusear a interface do framework através dos arquivos JSON e a flexibilidade do ambiente, que permite a adição/remoção de elementos na rede emulada.

Por fim, no capítulo 5, validou-se o framework MINITY através de comparação com com o framework descrito em (JAEGER et al., 2019). Pode-se averiguar que MINITY obteve êxito em encontrar resultados semelhantes, como a tardia percepção de alteração do RTT e da banda disponível do gargalo medido pelo TCP BBR. Ademais, concluiu-se que o BBR favorece, em relação a disponibilidade de banda, fluxos com maiores RTTs. Estes resultados corroboram com o que foi observado em (JAEGER et al., 2019), mas demostram uma enorme versatilidade em permitir cenários de rede muito mais complexos e de fácil configuração.

A grande contribuição deste trabalho é o desenvolvimento de uma ferramenta capaz de emular um ambiente de rede real no qual é possível testar diversos protocolos TCP frente a diversos cenários. A partir dos resultados de MINITY é possível realizar uma análise de sensibilidade na escolha de parâmetros de configuração dos protocolos e encontrar características de operação e de equidade frente ao compartilhamento de gargalo por múltiplos fluxos com diversos RTT. Sem a automação e flexibilidade de MINITY, a geração de resultados para ajuste de sensibilidade pode ser uma tarefa extremamente árdua.

Para desenvolvimento do framework MINITY foi utilizada a linguagem Python com o paradigma da orientação a objetos e o código contém, aproximadamente, 2000 linhas de código. É composto de 10 classes com 21 métodos e 30 funções auxiliares. Ademais, o seu funcionamento é separado em dois módulos com objetivos claros e distintos, o Handler e o Analyser. Por fim, sua produção envolveu a integração de 10 ferramentas diferentes, que estão disponibilizadas a partir de pacotes no LINUX e em bibliotecas externas para Python.

Para trabalhos futuros podem ser agregadas novas funcionalidades para tornar possível o teste de diversos outros protocolos, não apenas os da camada de transporte, mas também da camada de aplicação. Por exemplo, a implementação da habilitação de um servidor WEB no framework MINITY viabiliza testar os protocolos HTTP/2 (BELSHE; PEON; THOMSON, 2015) e validar o desenvolvimento do HTTP/3 (BISHOP, 2021). Ademais, com adição de um servidor WEB ao framework MINITY, o protocolo QUIC - da camada de transporte - de uso com HTTP/3, poderá ser facilmente estressado em diversos cenários.

O repositório do projeto MINITY pode ser encontrado em (LUNA, R. S., 2020), para a replicação de qualquer experimento ou alteração no código, o usuário deve clonar o projeto ou baixá-lo. Na página inicial há um breve manual de instalação dos pacotes utilizados por MINITY.

REFERÊNCIAS

MARK G.,ALEX W. *Iperf*. 2008. <https://iperf.fr/>.

WONDER NETWORK. *Global Ping Statistics*. 2021. <https://wondernetwork.com/pings>.

ALEXEY N. KUZNETSOV. **Utilitário de sockets**. 2001. <https://man7.org/linux/man-pages/man8/ss.8.html>.

ALEXEY N. KUZNETSOV, BERT HUBERT . **Configurações de controle de tráfego**. 2001. <https://linux.die.net/man/8/tc>.

ALLMAN, M.; PAXSON, V.; BLANTON, E. **TCP Congestion Control**. [S.l.], 2009. <http://www.rfc-editor.org/rfc/rfc5681.txt>. Disponível em: <http://www.rfc-editor.org/rfc/rfc5681.txt>.

BELSHE, M.; PEON, R.; THOMSON, M. **Hypertext Transfer Protocol Version 2 (HTTP/2)**. RFC Editor, 2015. RFC 7540. (Request for Comments, 7540). Disponível em: <https://rfc-editor.org/rfc/rfc7540.txt>.

BISHOP, M. **Hypertext Transfer Protocol Version 3 (HTTP/3)**. [S.l.], 2021. Work in Progress. Disponível em: <https://datatracker.ietf.org/doc/html/draft-ietf-quic-http-34>.

BRANDON HELLER. **MININET**. 2013. <http://mininet.org/api/index.html>.

BRAY, T. **The JavaScript Object Notation (JSON) Data Interchange Format**. [S.l.], 2017.

BROWN, P. Resource sharing of tcp connections with different round trip times. In: IEEE. **Proceedings ieee infocom 2000. conference on computer communications. nineteenth annual joint conference of the ieee computer and communications societies (cat. no. 00ch37064)**. [S.l.], 2000. v. 3, p. 1734–1741.

CARDWELL, N. et al. Bbr congestion control. **Working Draft, IETF Secretariat, Internet-Draft draft-cardwell-iccrg-bbr-congestion-control-00**, 2017.

CARDWELL, N. et al. Bbrv2: A model-based congestion control. In: **Presentation in ICCRG at IETF 104th meeting**. [S.l.: s.n.], 2019.

COMER, D. E. **Redes de Computadores e Internet-6**. [S.l.]: Bookman Editora, 2016.

DEMICHELIS, C.; CHIMENTO, P. **IP Packet Delay Variation Metric for IP Performance Metrics (IPPM)**. [S.l.], 2002. <http://www.rfc-editor.org/rfc/rfc3393.txt>. Disponível em: <http://www.rfc-editor.org/rfc/rfc3393.txt>.

Edgar Jamhour. **Tutorial QoS Linux**. 2015. <https://github.com/RodrigoSLuna/Minity>. Online; Acesso em 06 Outubro 2021.

GEORGE STAIKOS. **Configurador Kernel do Linux**. 1999. <https://www.linux.org/docs/man8/tc-netem.html>.

GETTYS, J. Bufferbloat: Dark buffers in the internet. **IEEE Internet Computing**, IEEE, v. 15, n. 3, p. 96–96, 2011.

GIAMPAOLO RODOLA. **Library FTP Server**. 2008. <https://pypi.org/project/pyftpdlib/>.

GUIDO VAN ROSSUM. **Library FTP User**. 1992. <https://docs.python.org/3/library/ftplib.html>.

HA, S.; RHEE, I.; XU, L. Cubic: a new tcp-friendly high-speed tcp variant. **ACM SIGOPS operating systems review**, ACM New York, NY, USA, v. 42, n. 5, p. 64–74, 2008.

HENDERSON, T. et al. **The NewReno Modification to TCP's Fast Recovery Algorithm**. [S.l.], 2012. <http://www.rfc-editor.org/rfc/rfc6582.txt>. Disponível em: <http://www.rfc-editor.org/rfc/rfc6582.txt>.

HOEILAND-JOERGENSEN, T. et al. **The Flow Queue CoDel Packet Scheduler and Active Queue Management Algorithm**. [S.l.], 2018.

HUBERT, B. et al. Linux advanced routing & traffic control howto. **Netherlabs BV**, v. 1, 2002.

HUNBERT. **TCLINUX**. 2001. <https://man7.org/linux/man-pages/man8/tc.8.html>.

HUNTER, J. D. Matplotlib: A 2d graphics environment. **Computing in Science & Engineering**, IEEE COMPUTER SOC, v. 9, n. 3, p. 90–95, 2007.

JACOBSON, V. Congestion avoidance and control. **ACM SIGCOMM computer communication review**, ACM New York, NY, USA, v. 18, n. 4, p. 314–329, 1988.

JAEGER, B. et al. Reproducible measurements of tcp bbr congestion control. **Computer Communications**, Elsevier, v. 144, p. 31–43, 2019.

JAFFE, J. Flow control power is nondecentralizable. **IEEE Transactions on Communications**, v. 29, n. 9, p. 1301–1306, 1981.

KAUR, K.; SINGH, J.; GHUMMAN, N. Mininet as software defined networking testing platform. In: . [S.l.: s.n.], 2014.

KAUR, K.; SINGH, J.; GHUMMAN, N. S. Mininet as software defined networking testing platform. In: **International Conference on Communication, Computing & Systems (ICCCS)**. [S.l.: s.n.], 2014. p. 139–42.

KLEINROCK, L. Power and deterministic rules of thumb for probabilistic problems in computer communications. In: **Proceedings of the International Conference on Communications**. [S.l.: s.n.], 1979. v. 43, p. 1–43.

KUROSE, K. R. J. **Redes de Computadores e a internet**. Brasil: PEARSON, 2013.

LAKSHMAN, T.; MADHOW, U. The performance of tcp/ip for networks with high bandwidth-delay products and random loss. **IEEE/ACM Transactions on Networking**, v. 5, n. 3, p. 336–350, 1997.

LANTZ, B.; HELLER, B.; MCKEOWN, N. A network in a laptop: rapid prototyping for software-defined networks. In: **Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks**. [S.l.: s.n.], 2010. p. 1–6.

LUNA, R. S. **Repositório MINITY**. 2020. <https://github.com/RodrigoSLuna/Minity>. Online; Acesso em 06 Setembro 2021.

MA, S. et al. Fairness of congestion-based congestion control: Experimental evaluation and analysis. **arXiv preprint arXiv:1706.09115**, 2017.

NAGLE, J. Congestion control in ip/tcp internetworks. **ACM SIGCOMM Computer Communication Review**, ACM New York, NY, USA, v. 14, n. 4, p. 11–17, 1984.

POSTEL, J. **User Datagram Protocol**. [S.l.], 1980. <http://www.rfc-editor.org/rfc/rfc768.txt>. Disponível em: <http://www.rfc-editor.org/rfc/rfc768.txt>.

POSTEL, J. **Transmission Control Protocol**. [S.l.], 1981. <http://www.rfc-editor.org/rfc/rfc793.txt>. Disponível em: <http://www.rfc-editor.org/rfc/rfc793.txt>.

POSTEL, J.; REYNOLDS, J. **File Transfer Protocol**. [S.l.], 1985. <http://www.rfc-editor.org/rfc/rfc959.txt>. Disponível em: <http://www.rfc-editor.org/rfc/rfc959.txt>.

RILEY, G. F.; HENDERSON, T. R. The ns-3 network simulator. In: WEHRLE, K.; GüNES, M.; GROSS, J. (Ed.). **Modeling and Tools for Network Simulation**. Springer, 2010. p. 15–34. ISBN 978-3-642-12330-6. Disponível em: <http://dblp.uni-trier.de/db/books/collections/Wehrle2010.html#RileyH10>.

STEPHEN HEMMINGER. **Emulador de Rede**. 2005. <https://www.linux.org/docs/man8/tc-netem.html>.

STOICA, I. **A Comparative Analysis of TCP Tahoe, Reno, New-Reno, SACK and Vegas**. [S.l.]: University of California Berkeley, CA, 2005.

SZWARCFITER, J. L. **Grafos e Algoritmos Computacionais**. Brasil: Editora Campus, 1983.

TANENBAUM, A. S. **Redes de Computadores**. trad. 5 ed. Rio de Janeiro: Pearson, 2011.

TEAM, T. pandas development. **pandas-dev/pandas: Pandas**. Zenodo, 2020. Disponível em: <https://doi.org/10.5281/zenodo.3509134>.

University of South California. **NETWORK TOOLS AND PROTOCOLS**. 2019. <http://ce.sc.edu/cyberinfra/workshops/Material/NTP/Lab%205.pdf>. Online; Acesso em 06 Outubro 2021.

Van Jacobson, Craig Leres and Steven McCanne. **Coletor de tráfego da rede**. 2010. <https://linux.die.net/man/8/tcpdump>.

WANG, R. et al. Tcp startup performance in large bandwidth networks. In: IEEE. **IEEE INFOCOM 2004**. [S.l.], 2004. v. 2, p. 796–805.

XU, L.; HARFOUSH, K.; RHEE, I. Binary increase congestion control (bic) for fast long-distance networks. In: IEEE. **IEEE INFOCOM 2004**. [S.l.], 2004. v. 4, p. 2514–2524.

APÊNDICES

APÊNDICE A – SIMULAÇÃO DE BUFFERBLOAT.

No exemplo proposto será averiguada a sensibilidade dos protocolos frente ao dimensionamento exagerado do buffer do gargalo a fim de verificar quais os protocolos são mais propensos a sofrerem com o problema do *Bufferbloat* (GETTYS, 2011). Este experimento consiste em configurar, erroneamente, o tamanho dos buffers de uma conexão, de maneira que o protocolo opere em uma faixa de banda maior do que a disponível, preenchendo as filas da conexão, e, consequentemente, ocasionando aumento da latência, do jitter e reduzindo a taxa de transferência global da rede.

Para este exemplo será utilizada uma topologia com 2 hospedeiros, um servidor e um cliente, conectados através de um comutador. A taxa de transmissão será diferente entre os enlaces a fim de garantir um único gargalo de conexão. Desta maneira, no enlace do servidor para o comutador a taxa de transmissão será de 20 Mbit/s, enquanto no enlace do comutador para o cliente será de 10 Mbit/s. Além disso, o RTT utilizado foi de 40 ms e o experimento consistirá de uma transferência ao longo de 100 segundos.

No framework MINITY o controle de tráfego nas interfaces é implementado a partir de duas filas, uma para o controle de banda e outra para adição de latência, jitter e enfileiramento dos pacotes excedentes, como visto no capítulo 3. A partir da observação das Figuras 32 e 33 pode-se inferir que pacotes estavam sendo descartados devido ao controle de banda exercido por MINITY. Para contornar este problema é necessário reconfigurar as interfaces de redes para garantir a existência de uma única disciplina de fila (*qdisc*). Devido à interatividade de MINITY com o usuário e a sua flexibilidade na configuração dos parâmetros de rede, pode-se realizar esta alteração. A partir da execução do método especificado na Figura 42, pode-se utilizar o *xterm* para acessar o utilitário tc e alterar as configurações de uma interface específica.

```
run(cli=True,iperf=False)
```

Figura 42 – Método run sendo executado com o parâmetro de abertura da interface do terminal para realizar a configuração manual da interface do gargalo.

Após a execução do método run será possível acessar a linha de comando das respectivas máquinas virtuais dos hospedeiros através da execução do comando: "xterm nome_hospedeiro". Desta maneira, uma janela com a linha de comando do respectivo hospedeiro se abrirá e será possível utilizar o utilitário tc para realizar as alterações necessárias. Assim, configuram-se as disciplinas das filas do hospedeiro servidor e do comutador através dos códigos encontrados na Figura 43. Observa-se que a alteração realizada no

hospedeiro "h1"configurou a fila de saída para garantir que a taxa de envio da banda seja de 20 Mbit/s, além de configurar a capacidade de armazenamento para 5 mil pacotes, ou seja, 7.5 MB. Já no comutador, alterou-se a configuração para garantir que a taxa de envio de banda seja de 10 Mbit/s e com uma capacidade de 15 mil pacotes, ou seja, 22.5 MB. Em ambas as configurações não foram adicionadas latência para a transmissão dos pacotes. Nota-se que o BDP desta transferência será de 0.05 MB, sendo recomendado ajustar as capacidades das filas para 2*BDP, 0.1 MB. Assim, a fila do gargalo estará ajustada com um valor 225 vezes maior do que o recomendado. Esse superdimensionamento da capacidade da fila do gargalo tem como objetivo demonstrar a ocorrência do *Bufferbloat*.

```
tc qdisc del dev h1-sw1 root
tc qdisc add dev h1-sw1 root handle 5:0 netem rate 20Mbit limit 5k delay 0ms
```

(a) No primeiro comando, remove-se a configuração padrão de MINITY para a interface. No segundo configura-se a disciplina de uma fila na interface, com o NetEM (STEPHEN HEMMINGER, 2005) para limitar a taxa de tranmissão em 20 Mbit/s e adicionar uma capacidade de armazenamento de 5 mil pacotes e com 0 de atraso na fila de saída da interface de rede.

```
tc qdisc del dev sw1-h2 root
tc qdisc add dev sw1-h2 root handle 5:0 netem rate 10Mbit limit 15k delay 0ms
```

(b) No primeiro comando, remove-se a configuração padrão de MINITY para a interface. No segundo configura-se a disciplina de uma fila na interface, com o NetEM (STEPHEN HEMMINGER, 2005) para limitar a taxa de tranmissão em 10 Mbit/s e adicionar uma capacidade de armazenamento de 15 mil pacotes e com 0 de atraso na fila de saída da interface de rede.

Figura 43 – Comandos utilizados para a configuração do controle de tráfego nas interfaces do servidor e do comutador da conexão.

O experimento foi executado para três protocolos TCP: 1) BBR; 2) NewRENO; e 3) CUBIC. Os resultados podem ser observados através das Figuras 44, 45 e 46, respectivamente. Foram gerados gráficos referentes à taxa de envio no servidor, à ocupação da fila de saída do servidor e à ocupação da fila do comutador - gargalo da conexão. Observa-se que, apesar do dimensionamento da capacidade das filas no gargalo ser de 22.5 MB, o limite alcançado, como pode ser visto nas Figuras 45c e 46c, foi de aproximadamente 8 MB, valor máximo de memória disponilizado pela interface física de rede utilizada.

Na Figura 44 são apresentados os resultados obtidos para o protocolo TCP BBR. Pode-se observar que o protocolo logrou êxito ao evitar a ocupação máxima da fila do gargalo da conexão. Durante a transmissão, o valor ocupado na fila do gargalo manteve-se com, no máximo, 0.02 MB, ou seja, 2*BDP da conexão. Ademais, devido à correta medição da banda do gargalo e do RTT da conexão, a taxa de banda efetiva durante a transferência se manteve próximo do valor limítrofe do gargalo, de 10 Mbit/s.

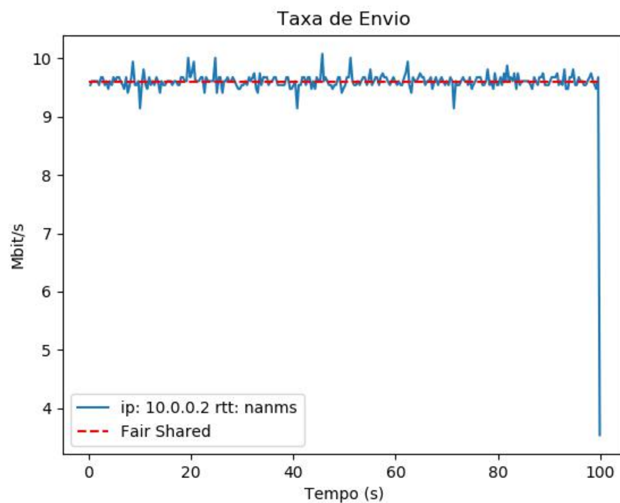
Já na Figura 45 observa-se os resultados do experimento com o TCP NewRENO. Devido ao seu controle de congestionamento ser baseado em perdas de pacotes, o protocolo mantém a transferência em torno de 20 Mbit/s nos primeiros 9 segundos. Após este momento ocorre uma perda devido ao preenchimento da fila do gargalo da conexão, como visto na Figura 45c. Assim, o protocolo reage à ocorrência de perda, reduzindo a taxa de transferência pela metade. É importante ressaltar que os resultados da taxa de envio

envolvem a quantidade média de pacotes que transitam pela interface de saída em um intervalo de tempo pré-definido pelo usuário, neste experimento utilizou-se o valor de 0.09 segundos. Após o preenchimento da fila do gargalo, com uma quantidade de pacotes enfileirados acima de 8 MB, ocorre a redução pela metade da taxa de envio em intervalos de 6 segundos, aproximadamente. Este comportamento pode ser observado a partir do instante de 9 segundos, com o esvaziamento da fila de saída do servidor, visto na Figura 45b. Também pode-se observar na Figura 45a a redução de banda. A partir desses eventos, pode-se inferir que o protocolo está recuperando da perda de pacotes. Os intervalos, com repetição ao longo do experimento, são decorrentes do RTT real da conexão ser em torno de 6 segundos, ao invés dos 40 ms configurados no experimento. Ademais, a taxa de transferência após o preenchimento da fila do gargalo da conexão se mantém com valor superior ao disponibilizado, próximo de 18 Mbit/s. A errônea configuração das filas nos equipamentos de rede em uma conexão pode ocasionar o aumento no RTT, que é um indicativo da ocorrência do *Bufferbloat* (GETTYS, 2011) na conexão. E, ainda, como o protocolo opera acima do ponto de operação ótimo, *Kleinrock Point* (KLEINROCK, 1979), mantém-se a máxima utilização a taxa de ocupação da fila do gargalo.

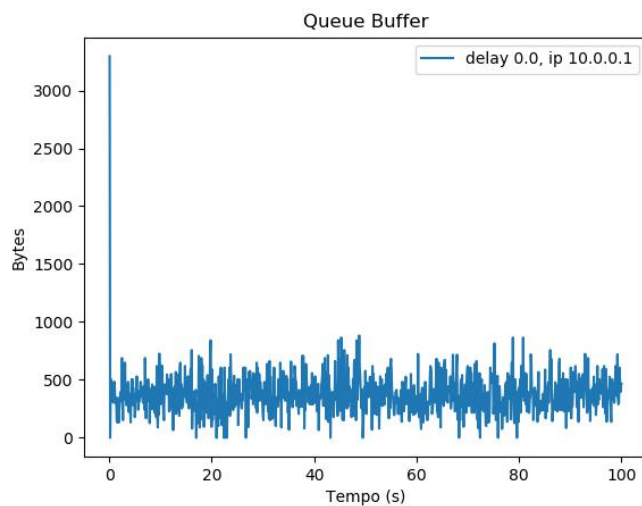
Por fim, na Figura 46 encontram-se os resultados do experimento com o TCP CUBIC. Nos primeiros 9 segundos de transferência pode-se observar um comportamento diferenciado no crescimento da quantidade de bytes enfileirados na fila do gargalo e na taxa de envio do protocolo, quando comparado ao intervalo de tempo entre os instantes 10 e 25 segundos. Esse comportamento referente ao início da transferência do TCP CUBIC caracteriza o comportamento do algoritmo slow-start utilizado pelo protocolo. Após a fase de slow-start, o protocolo entra na fase de controle de congestionamento e, até o preenchimento da fila do gargalo da conexão, o protocolo utiliza a parte convexa da função cúbica para encontrar um novo ponto ótimo de operação. Este comportamento ocorre até o momento do preenchimento completo da fila do gargalo. Após, ocorre uma de perda de pacote, no instante de aproximadamente 25 segundos, como pode ser visto nas Figuras 46b e 46c. A taxa de transferência é reduzida de 15 Mbit/s para aproximadamente 12 Mbit/s, uma redução de 25%, valor referente ao parâmetro utilizado pelo protocolo na função cúbica do algoritmo de controle de congestionamento. Portanto, pode-se observar que, como controle de congestionamento do TCP CUBIC é baseado em perda de pacote, o errôneo dimensionamento da capacidade da fila pode ocasionar o *Bufferbloat* (GETTYS, 2011) devido à ocupação máxima da fila do gargalo da conexão, acarretando em uma maior latência durante a transmissão.

Os resultados obtidos pelo experimento realizado através do framework MINITY demonstram a superioridade do protocolo TCP BBR quando comparado ao TCP CUBIC e TCP NewRENO, apresentando a característica de manter uma menor pressão na fila do gargalo, evitando-se a ocorrência do *Bufferbloat.* Independente do dimensionamento da capacidade da fila, o protocolo manteve uma taxa de ocupação em torno de 2*BDP

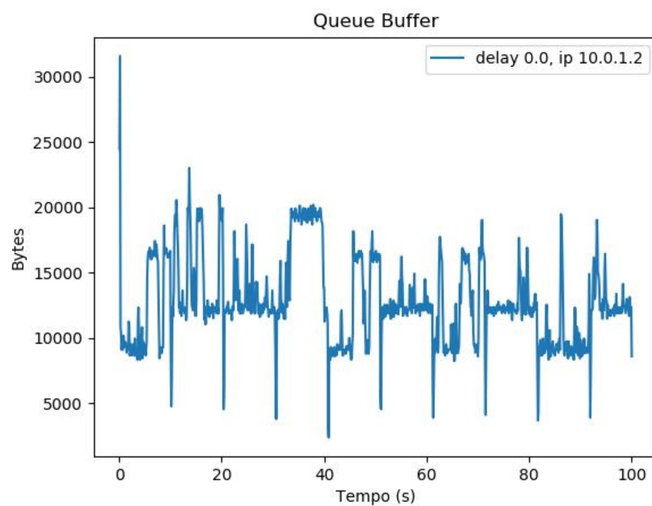
da conexão. Portanto, pode-se afirmar que o framework MINITY pode ser utilizado para analisar os comportamentos dos diversos protocolos de controle de congestionamento disponíveis. Ademais, a sua exatidão em capturar as características inerentes a cada protocolo demonstra a sua flexibilidade para analisar os protocolos em diversos cenários. Após diversos experimentos e comparações dos resultados obtidos por MINITY com outros frameworks, e com o comportamento esperado dos protocolos, pode-se afirmar que MINITY é um framework promissor, capaz de contribuir para a experimentação, prototipação e análise de diversos protocolos.



(a) Taxa de envio do servidor durante o experimento.

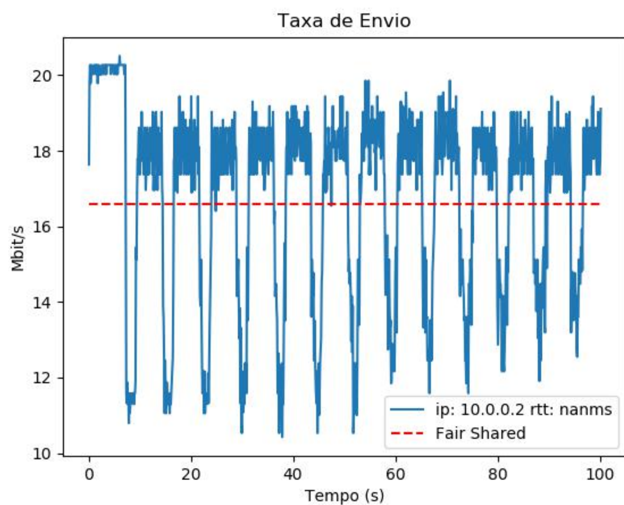


(b) Quantidade de bytes enfileirados na fila do servidor da conexão.

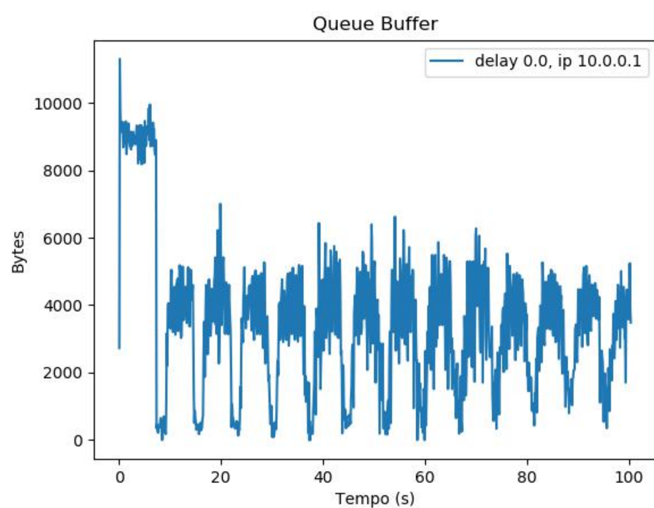


(c) Quantidade de bytes enfileirados no comutador da conexão.

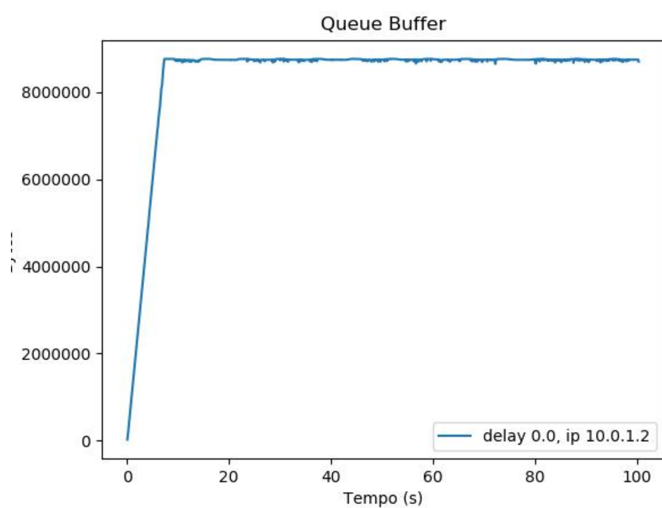
Figura 44 – Resultados do experimento ao utilizar o protocolo BBR. Em (a) observa-se a taxa de envio, referente ao valor médio da transferência da conexão em um intervalo de 0.08 segundos. Já em (b) a figura refere-se a quantidade de bytes na fila de saída do servidor. Por fim, em (c) o resultado da fila do gargalo da conexão, extraído do comutador.



(a) Taxa de envio do servidor durante o experimento.

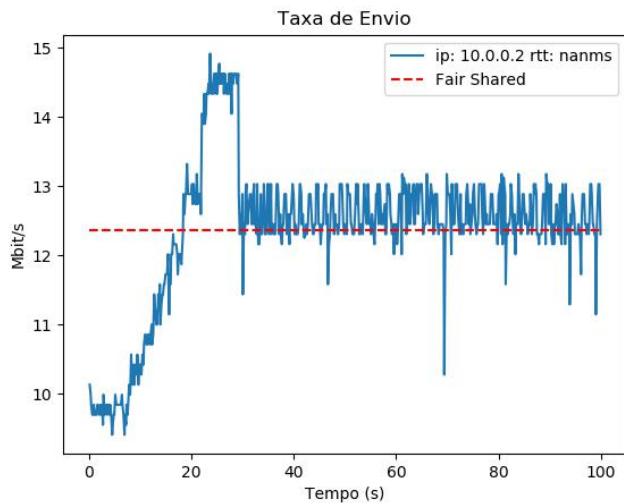


(b) Quantidade de bytes enfileirados na fila do servidor da conexão.

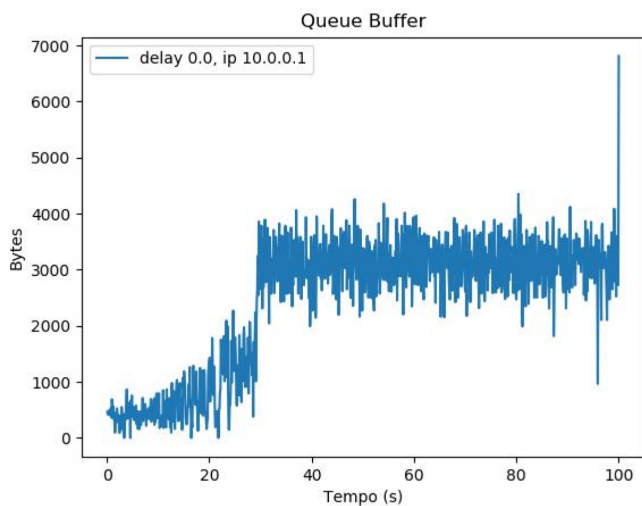


(c) Quantidade de bytes enfileirados no comutador da conexão.

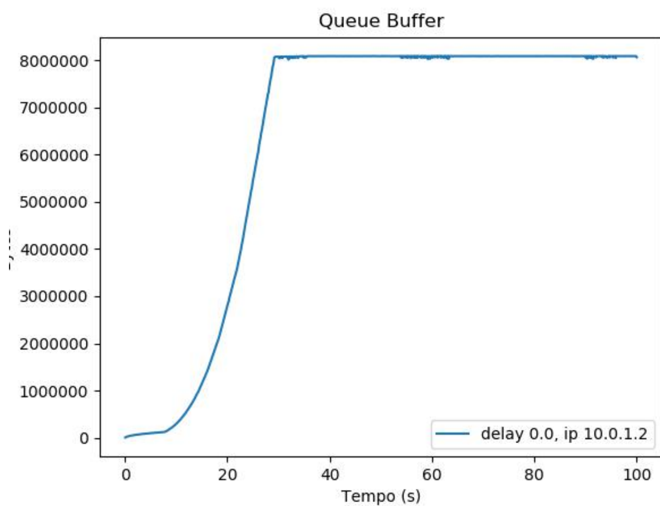
Figura 45 – Resultados do experimento ao utilizar o protocolo NewRENO. Em (a) observa-se a taxa de envio, referente ao valor médio da transferência da conexão em um intervalo de 0.08 segundos. Já em (b) a figura refere-se a quantidade de bytes na fila de saída do servidor. Por fim, em (c) o resultado da fila do gargalo da conexão, extraído do comutador.



(a) Taxa de envio do servidor durante o experimento.



(b) Quantidade de bytes enfileirados na fila do servidor da conexão.



(c) Quantidade de bytes enfileirados no comutador da conexão.

Figura 46 – Resultados do experimento ao utilizar o protocolo CUBIC. Em (a) observa-se a taxa de envio, referente ao valor médio da transferência da conexão em um intervalo de 0.08 segundos. Já em (b) a figura refere-se a quantidade de bytes na fila de saída do servidor. Por fim, em (c) o resultado da fila do gargalo da conexão, extraído do comutador.

APÊNDICE B – SIMULAÇÃO DE BUFFERBLOAT - GARGALO COM 6*BDP

Neste experimento será averiguado o comportamento dos protocolos TCP BBR, NewReno e CUBIC frente ao problema do bufferbloat (GETTYS, 2011). Este problema consiste em configurar o erroneamente o tamanho dos buffers de uma conexão de certa maneira que o protocolo opere em uma faixa de banda maior do que a disponível o que pode ocasionar aumento da latência e do jitter. Usualmente, recomenda-se que o gargalo da conexão seja da ordem de 2*BDP.

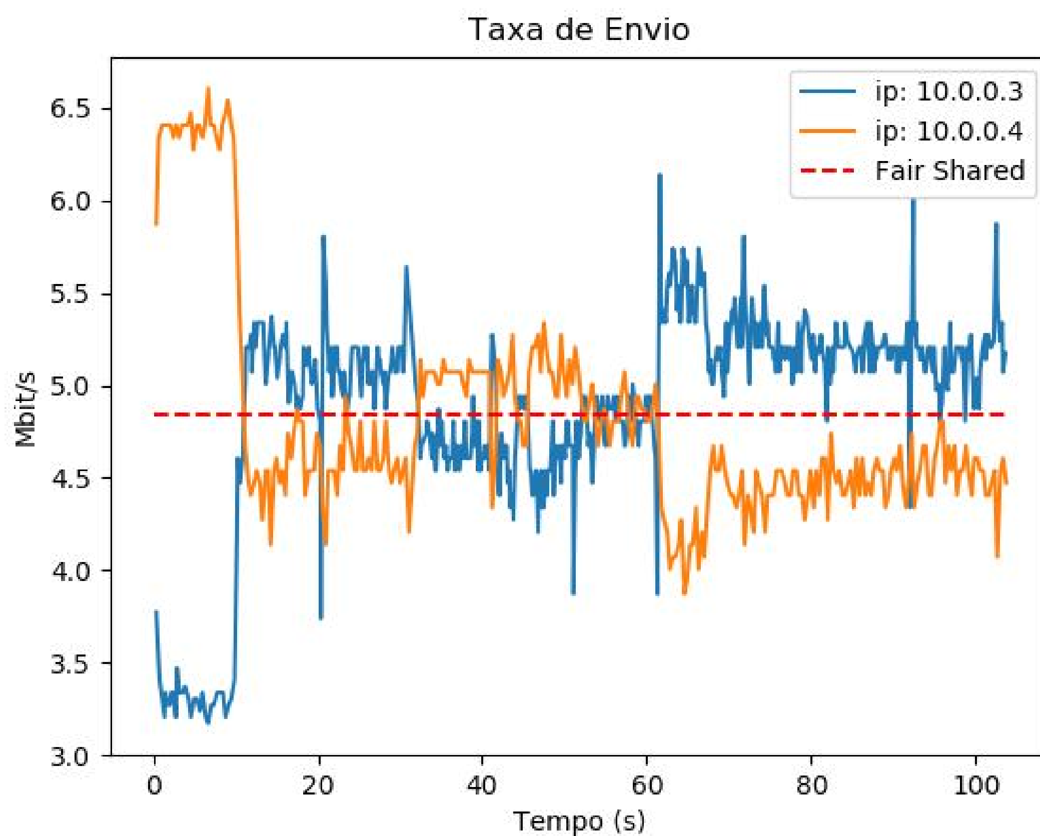
Para este exemplo será utilizado a topologia de dumbbell, encontrada na Figura 25, com N = 2 e uma taxa de banda do gargalo inicial com 10Mbit/s e dois fluxos, um com RTT de 10 ms e outro com RTT de 100 ms. O experimento consistirá de uma transferência durante 120 segundos e ao longo de sua execução a rede sofrerá alterações de banda no gargalo. Ademais o buffer do gargalo será redimensionado para que aceite a quantidade máxima de 500 pacotes, ou seja, 0.75 MB - considerando as condições iniciais da rede e o valor máximo de RTT - cerca de 3 vezes o valor recomendado para a capacidade do gargalo da conexão.

Os resultados obtidos podem ser encontrados nas Figuras 47, 48 e 49. Pode-se averiguar que o protocolo TCP BBR manteve uma pressão menor na fila do gargalo, operando em torno de 55000 bytes, ou seja, em aproximadamente 37 pacotes. Perceba que este valor de operação é da ordem de 10 vezes menor do que a capacidade do gargalo. Ademais, pode-se observar novamente o comportamento característico do TCP BBR em favorecer fluxos com menor RTT, comportamento inverso do TCP clássico. Apesar de uma diferença pouco significativa quando comparado ao comportamento do TCP clássico, o fluxo com 10 ms obteve em torno de 5.5 Mbit/s enquanto o fluxo com RTT de 100 ms, operou com 4.5 Mbit/s.

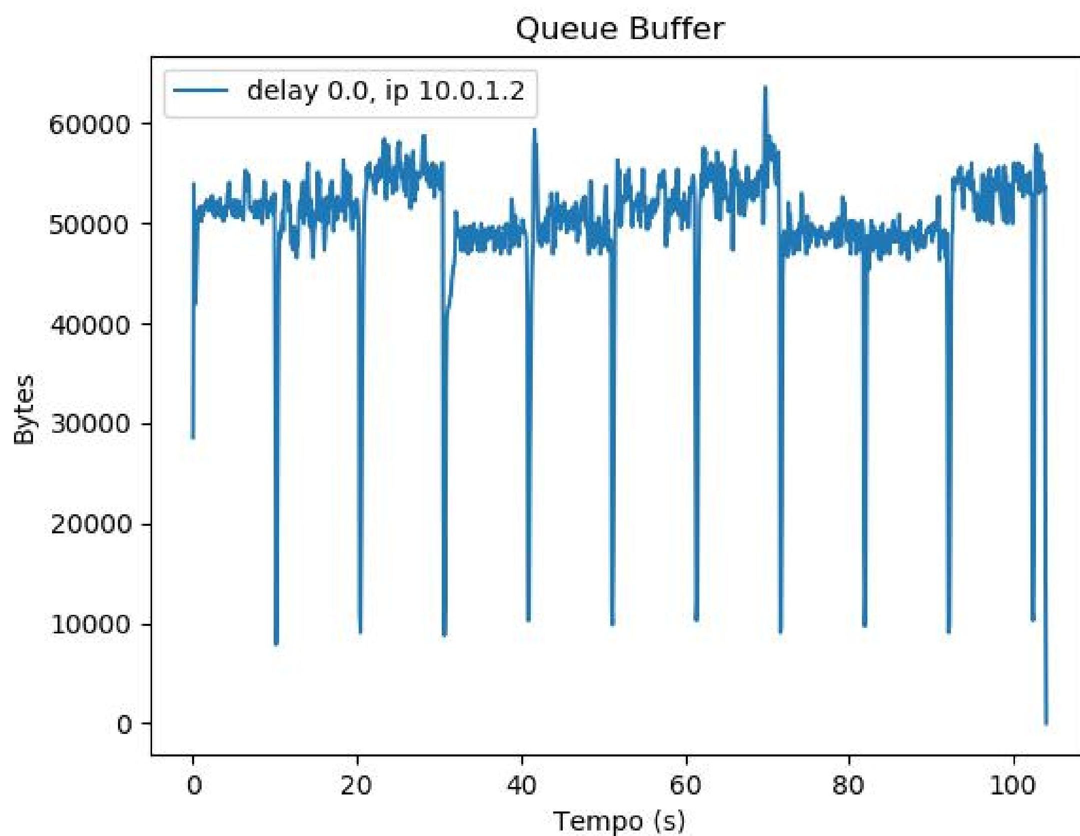
Ao observar o comportamento do protocolo TCP CUBIC, na Figura 48. Pode-se verificar que após preencher completamente o gargalo, no instante 10, a ocupação do gargalo é reduzida e nos instantes seguintes observa-se um comportamento recorrente da utilização da função de ajuste, tanto a sua parte côncava quanto a convexa.

Por fim, na Figura 49 encontra-se os resultados do experimento com TCP New Reno. Percebe-se que rapidamente o protocolo preenche a fila do gargalo da conexão. Isso se deve a fase slow-start do protocolo. O comportamento linear característico da fase de prevenção ao congestionamento pode ser observado a partir do instante 5.

Portanto, mais uma vez o framework MINITY pode ser útil para verificar a superioridade do protocolo TCP BBR em relação a ocupação de fila do gargalo frente aos protocolos TCP CUBIC e NewReno.

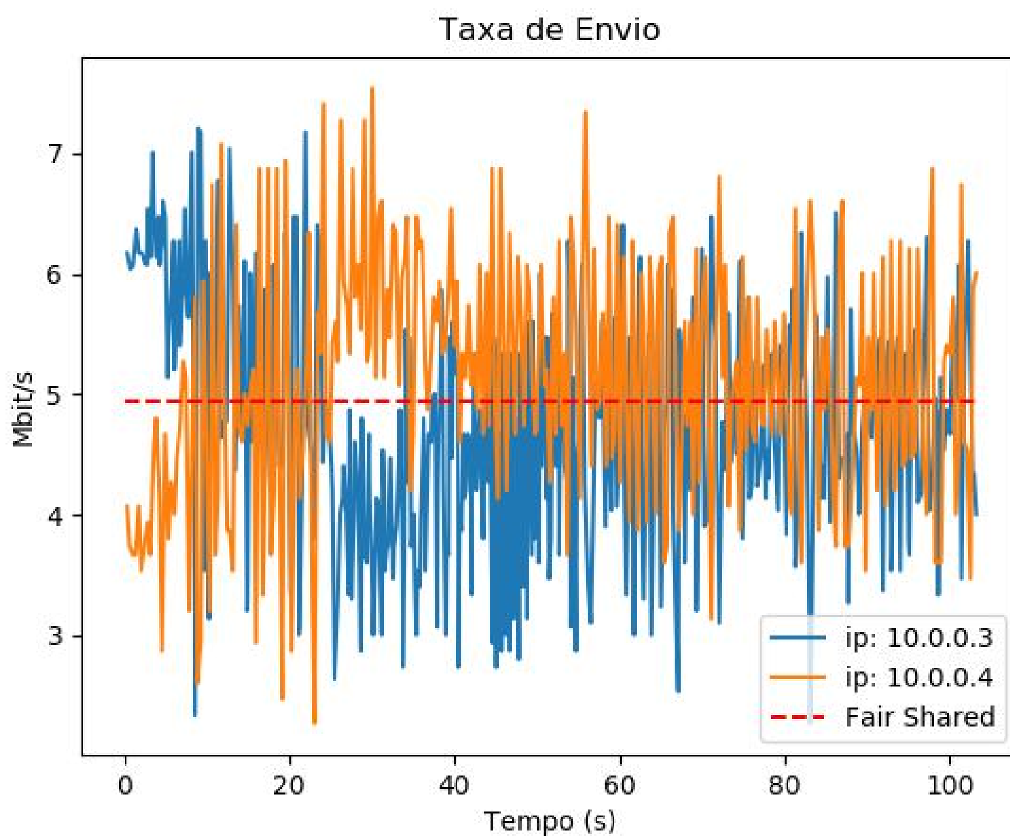


(a) Taxa de envio dos fluxos. Em azul o fluxo com experimenta um RTT de 10 ms. Já o fluxo em laranja experimenta um RTT de 100 ms.

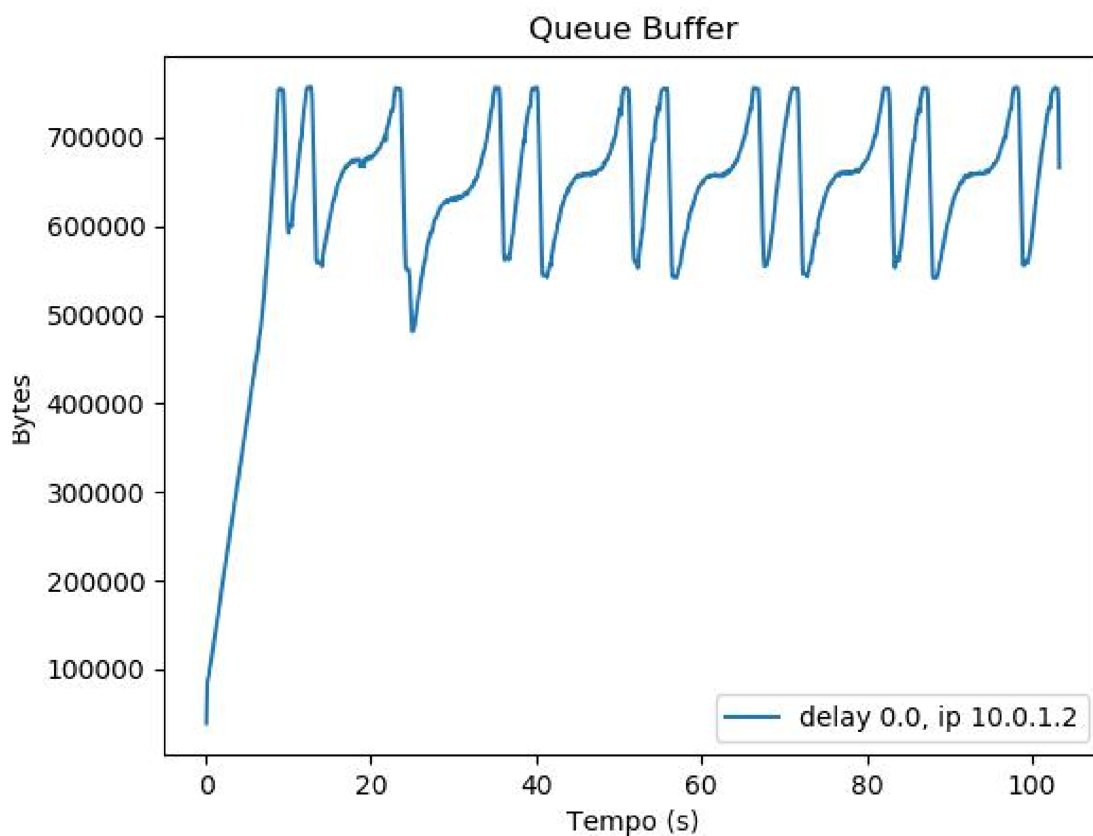


(b) Quantidade de bytes enfileirados no gargalo da conexão.

Figura 47 – Resultado do experimento com 2 fluxos TCP BBR compartilhando um gargalo com 10 Mbit/s. O tamanho do gargalo foi configurado para suportar 500 pacotes.

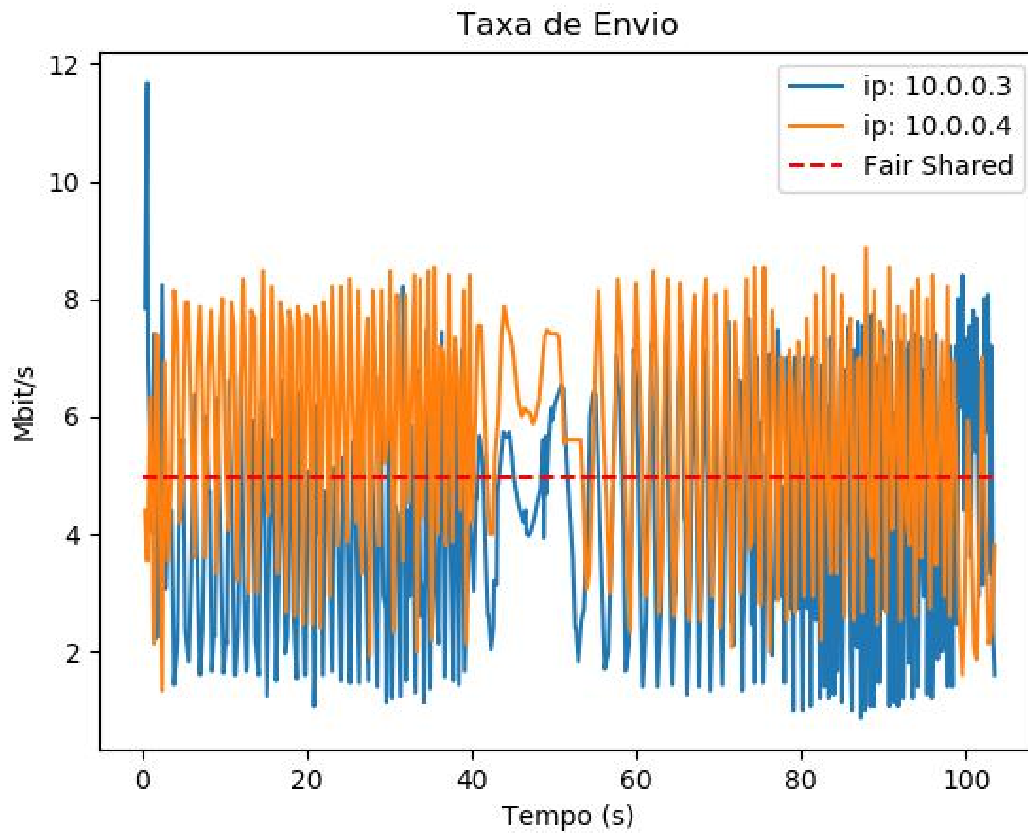


(a) Taxa de envio dos fluxos. Em azul o fluxo com experimenta um RTT de 10 ms. Já o fluxo em laranja experimenta um RTT de 100 ms.

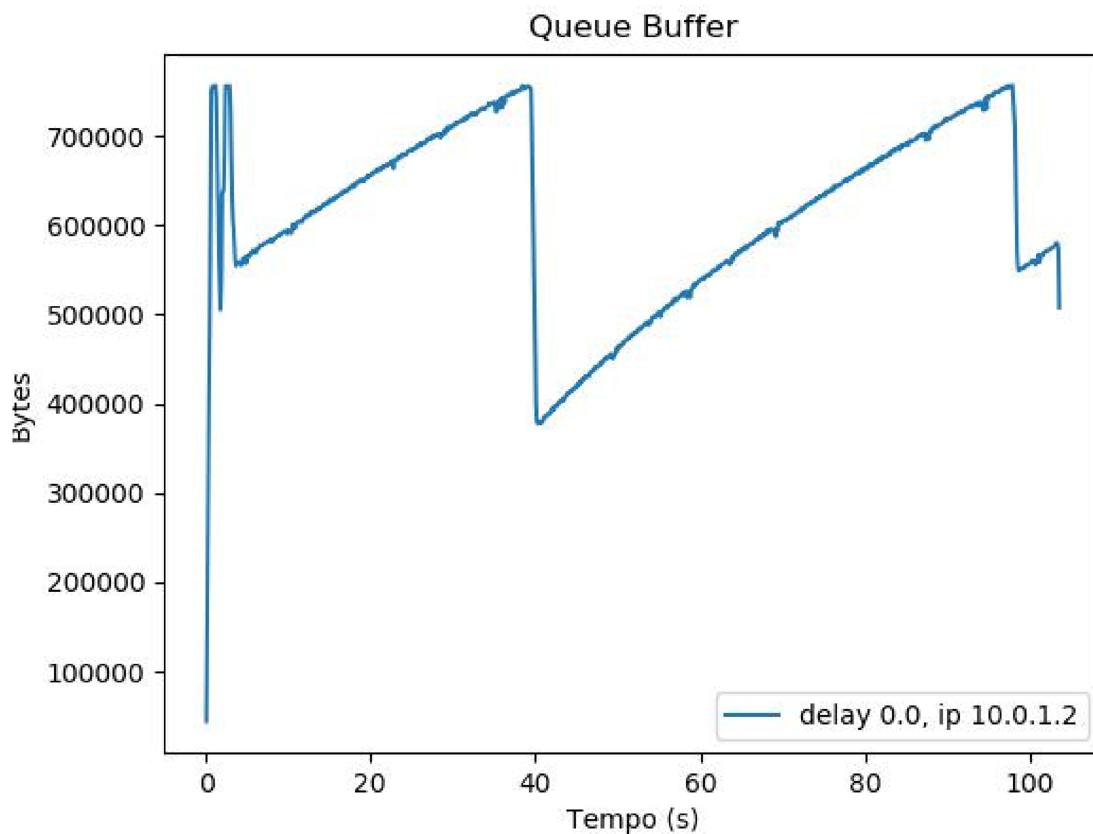


(b) Quantidade de bytes enfileirados no gargalo da conexão.

Figura 48 – Resultado do experimento com 2 fluxos TCP CUBIC compartilhando um gargalo com 10 Mbit/s. O tamanho do gargalo foi configurado para suportar 500 pacotes.



(a) Taxa de envio dos fluxos. Em azul o fluxo com experimenta um RTT de 10 ms. Já o fluxo em laranja experimenta um RTT de 100 ms.



(b) Quantidade de bytes enfileirados no gargalo da conexão.

Figura 49 – Resultado do experimento com 2 fluxos TCP RENO compartilhando um gargalo com 10 Mbit/s. O tamanho do gargalo foi configurado para suportar 500 pacotes.