Teleprocessamento e Redes - Relatório do trabalho final

Leonardo Ribeiro Santiago (120036072)
João Matheus Nascimento Gonçalves (120023786)
Esteves Emmanuel Melo Ferreira (117209640)

1 Introdução

Neste relatório iremos responder as perguntas relacionadas à parte 2 do trabalho final. O código está disponível no seguinte repositório do github: github.com/o-santi/redes.

Para reproduzir os resultados, deve-se instanciar uma máquina virtual Ubuntu usando Vagrant, assim como descrito em github.com/kaichengyan/mininet-vagrant. Uma vez dentro da VM, clonamos o repositório git para uma pasta interna, e rodamos o arquivo `run.sh`.

```
git clone -b entrega-preliminar https://github.com/o-santi/redes.git ~/redes
cd ~/redes/bufferbloat
chmod +x ./run.sh
sudo ./run.sh
```

Isto irá rodar os dois casos de teste (`max_queue=20` e `max_queue=100`) e gerar os gráficos citados neste relatório.

2 Parte 2

2.1 Qual é o tempo médio de busca da página da web e seu desvio padrão quando q=20 e q=100?

No caso `q=20`, o tempo médio de busca da página é de 3,30 segundos, com um desvio padrão de 0,82.

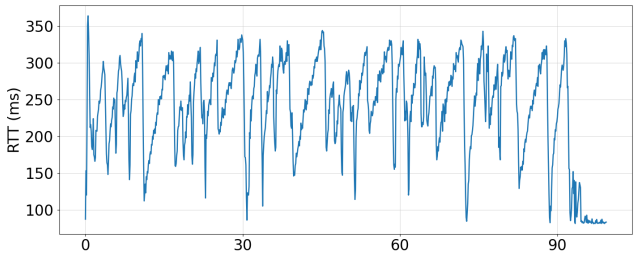

Figure 1: Tempo de resposta dos pings ao longo da duração do teste.

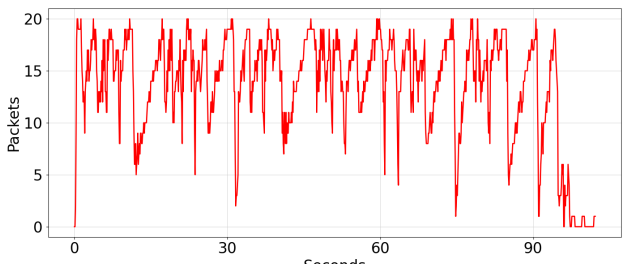

Figure 2: Número de pacotes na fila do switch ao longo do teste.

Já no caso `q=100`, o tempo médio é de 9,96 segundos, com um desvio padrão de 3,17.

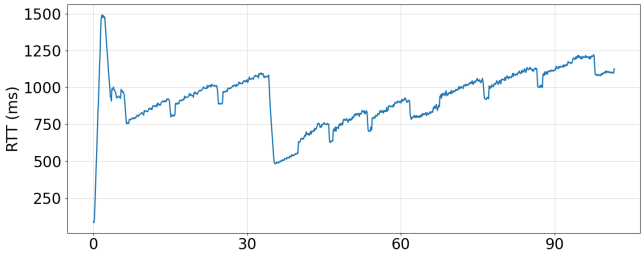

Figure 3: Tempo de resposta dos pings ao longo da duração do teste.

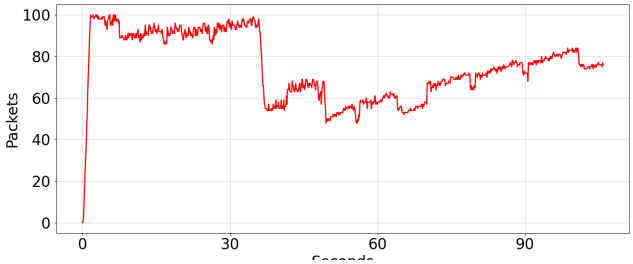

Figure 4: Número de pacotes na fila do switch ao longo do teste.

2.2 Por que você vê uma diferença nos tempos de busca de páginas da Web com buffers de roteador curtos e grandes?

A diferença pode ser explicada pela maior quantidade de pacotes que entram na fila, fazendo com que a janela de congestão da conexão TCP aumente, fazendo com que os pacotes passem mais tempo na fila esperando para serem transmitidos.

Ao diminuir o tamanho da fila para 20, os pacotes param de acumular como antes, fazendo com que o tempo que um pacote deve esperar na fila seja reduzido.

2.3 Bufferbloat pode ocorrer em outros lugares, como sua placa de interface de rede (NIC). Verifique a saída de ifconfig eth0 de sua VM mininet. Qual é o comprimento (máximo) da fila de transmissão na interface de rede relatada pelo ifconfig? Para esse tamanho de fila, se você assumir que a fila é "drenada" a 100 Mb/s, qual é o tempo máximo que um pacote pode esperar na fila antes de sair da NIC?

Rodando `h1 ifconfig` de dentro da mininet, vemos que o parâmetro `txqueuelen` da interface principal `h1-eth0` é de 1000 pacotes, com `mtu=1500 bytes`.

Isso significa que, se um pacote entrar na última posição da fila, ele deve esperar todos os 999 pacotes transmitirem, e depois esperar o tempo da sua própria transmissão. O tempo máximo de transmissão de um pacote, na velocidade de `100Mb/s` é de $\frac{1000*1500*8}{100.000.000}$ segundos $= \frac{15*8}{1000}$ segundos $\approx 0,12$ segundos.

2.4 Como o RTT relatado pelo ping varia com o tamanho da fila? Descreva a relação entre os dois.

Tanto no caso `q=20` quanto quando `q=100`, o `RTT` aumenta linearmente com o número de pacotes na fila. Isso se dá pois o tempo de transmissão na rede é constante, dado que o único gargalo é o switch principal. Assim, quanto mais pacotes na fila do switch, maior será o tempo que ele levará para ser transmitido, e portanto maior será o RTT.

2.5 Identifique e descreva duas maneiras de mitigar o problema de bufferbloat.

De modo geral, técnicas para mitigar o *bufferbloat* podem ser separadas em duas categorias: as que visam melhorar a rede e as que visam melhorar as pontas da conexão.

Dos que visam melhorar a rede, vale ressaltar os algorítmos `CoDel` (*Controled Delay*) e sua melhoria `FQ-Codel` (*Fair/Flow Queue CoDel*), que está dentro da categoria de algoritmos de *Active Queue Management* (`AQM`). Esse algoritmo busca controlar o limite do delay que os pacotes experienciam nas filas dos roteadores para um máximo de 5 millisegundos. Caso o número de pacotes aumente rapidamente, de forma que o delay passe desse *threshold*, pacotes são descartados da fila até que o delay esteja dentro do limite aceitável.

Dos que visam melhorar as pontas da conexão, destaca-se uma implementação do protocolo TCP utilizando um algoritmo de congestão diferente do `Reno`: o *Bottleneck Bandwidth and Round-trip propagation time* (`BBR`). Diferentemente do `Reno`, que utiliza a perda de pacotes para detectar congestionamento e baixas taixas de transmissão, o `BBR` constroi um modelo da rede, utilizandos amostras de pacotes para medir a taxa de transmissão e o *Round Trip Time* (`RTT`).

3 Parte 3

Para gerar os dados utilizando o método BBR, usamos o script `bufferbloat/run_bbr.sh`.

3.1 Qual é o tempo médio de busca da página da web e seu desvio padrão quando q=20 e q=100?

No caso `q=20`, o tempo médio de busca da página é de 2,47 segundos, com um desvio padrão de 1,27.

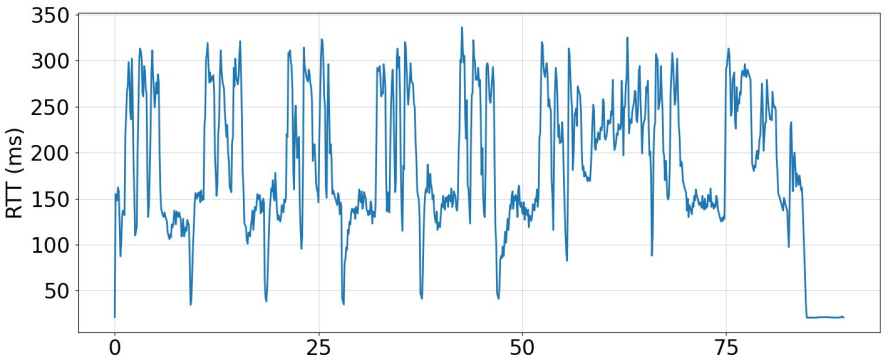

Figure 5: Tempo de resposta dos pings ao longo da duração do teste.

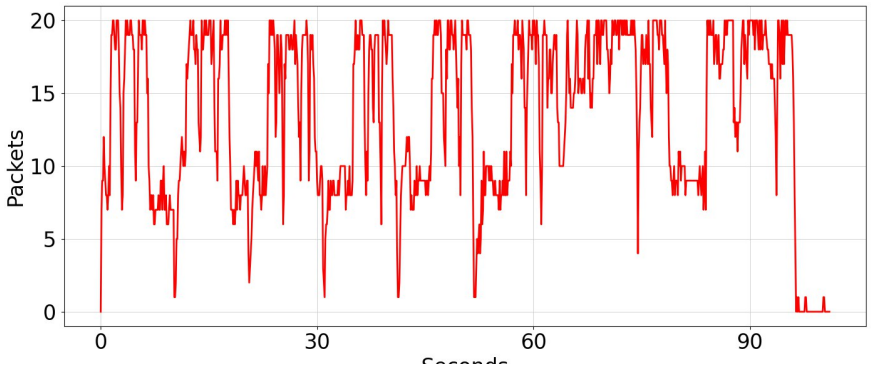

Figure 6: Número de pacotes na fila do switch ao longo do teste.

No caso `q=100`, o tempo médio de busca da página é de 1,78 segundos, com um desvio padrão de 0,05.

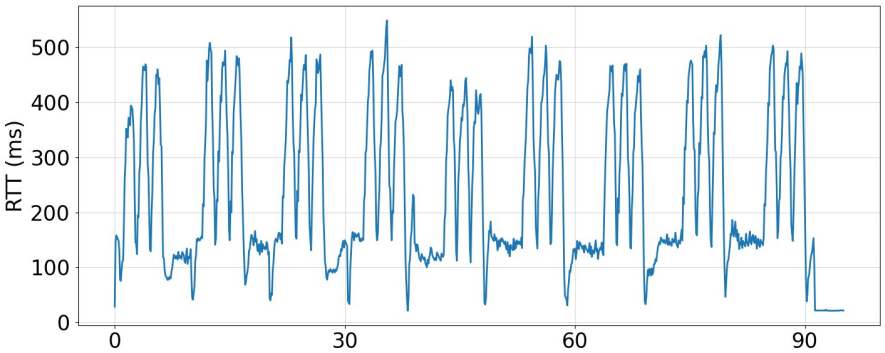

Figure 7: Tempo de resposta dos pings ao longo da duração do teste.

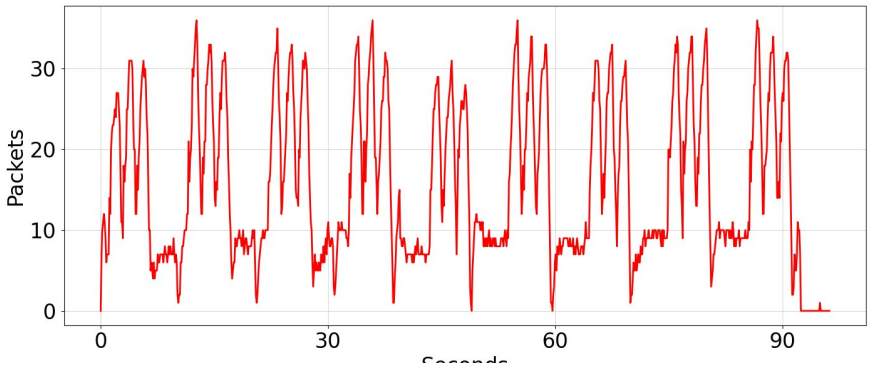

Figure 8: Número de pacotes na fila do switch ao longo do teste.

3.2 Compare o tempo de busca da página web entre q=20 e q=100 da Parte 3. Qual tamanho da fila fornece um tempo de busca menor? Como isso é diferente da Parte 2?

Vemos que o tempo quando `q=100` é consideravelmente menor do que quando `q=20` - aproximadamente 40% mais rápido. Esse resultado contrasta com o método de controle de congestão Reno, onde a fila maior causa tem um tempo médio 3 vezes maior.

3.3 Você vê a diferença nos gráficos de tamanho de fila da Parte 2 e da Parte 3? Dê uma breve explicação para o resultado que você vê.

No gráfico do método Reno, vemos que tanto o RTT quanto o tamanho do buffer tendem a sempre crescer, e nunca voltam a um estado "vazio". No caso do BBR, vemos que, ainda que o buffer fique cheio (no caso `q=20`) ele ainda sim tende a voltar para um estado vazio, sem crescimento indefinido. Por isso, vemos que o tempo médio de resposta no caso `q=100` do BBR não possui quase nenhuma variância, tendo um desvio padrão de apenas 0,05. É importante ressaltar também que o método BBR teve tempo de resposta médio consideravelmente menor, em ambos os casos testados.

3.4 Você acha que resolvemos o problema do bufferbloat? Explique seu raciocínio.

O caso de estudo utilizado é simplório demais para que possamos afirmar com certeza que o problema do bufferbloat é resolvido pelo algoritmo BBR, pois só considera um caso específico de fluxo contínuo de pacotes TCP, com latência não muito importante. Para determinar se o bufferbloat fora resolvido, seria

necessário analizar uma ampla gama de casos mais complexos, envolvendo sistemas com fluxo sensível à latência e com muito mais pacotes (por exemplo, *streaming* de vídeo ou acesso de muitas pessoas à internet).

Ainda sim, para o caso testado, vemos que o buffer nunca chega a ficar cheio sob o BBR no caso `q=100`, com um máximo de 35 a 40 pacotes na fila. Logo, podemos concluir que, para esse caso específico, o método TCP BBR resolve sim o bufferbloat, pois aumentar o buffer não implica em piora da qualidade do sistema - pelo contrário, significou melhora no tempo médio de resposta.

4 Parte 4

Para analizar o protocolo QUIC, algumas modificações foram feitas ao arquivo `bufferbloat.py`, para utilizar binários gerados pela implementação do protocolo do Cloudflare em Rust, chamada quiche. Assim, o arquivo `bufferbloat_quic.py` apresenta as seguintes modificações:

1. A implementação do servidor, que antes era feita rodando `python webserver.py`, passa a ser feita pelo binário `/redes/bin/quiche-server`. Para gerá-lo, clonamos o repositório do quiche, seguindo as instruções do repositório, e rodamos

   ```
   cd <quiche-repositorio>
   cargo build --bin quiche-server --target-dir \texttildelow/redes/bin
   ```

2. A implementação do cliente, que antes era feita rodando `curl`, passa a ser feita pelo binário `~/redes/bin/quiche-client`. Para gerá-lo, rodamos

   ```
   cd <quiche-repositorio>
   cargo build --bin quiche-client --target-dir \texttildelow/redes/bin
   ```

3. A implementação do `iperf` fora trocada pelo binário qperf, para medir a performance da rede, já que a ferramenta `iperf` não consegue, por padrão, entender o protocolo QUIC.

Ao rodar o arquivo modificado, geramos o seguinte gráfico para o caso `q=20`:

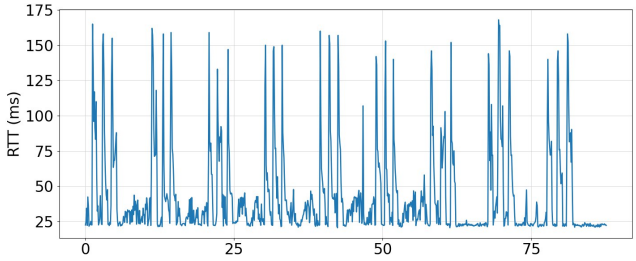

Figure 9: Tempo de resposta dos pings ao longo da duração do teste.

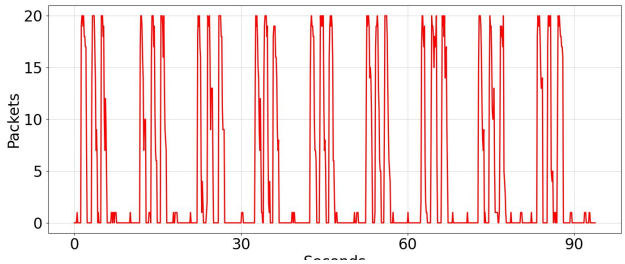

Figure 10: Número de pacotes na fila do switch ao longo do teste.

Vemos que o tempo médio de resposta é de 1,76 segundos, com desvio padrão de 0,15. Respectivamente, para o caso `q=100`:

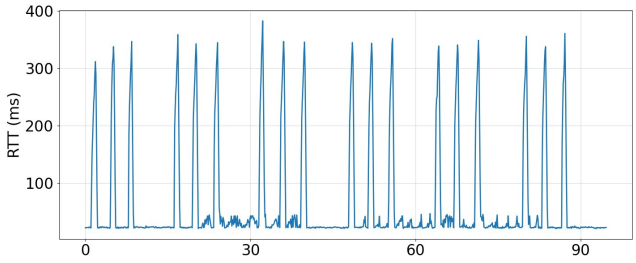

Figure 11: Tempo de resposta dos pings ao longo da duração do teste.

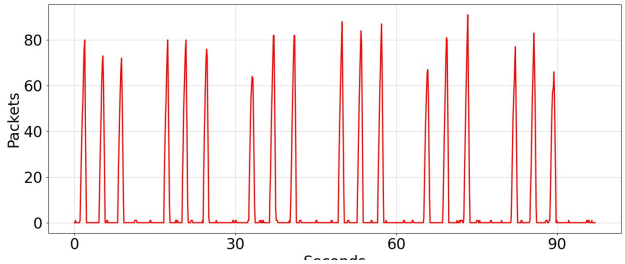

Figure 12: Número de pacotes na fila do switch ao longo do teste.

Vemos que o tempo médio de resposta é de 3,66 segundos, com desvio padrão de 0,17.

Notamos que, assim como o TCP Reno, o tamanho do buffer ser maior implicou em um tempo médio de resposta maior, ou seja, o bufferbloat ainda está presente. Apesar disso, vemos que as características dos gráficos são muito distintas, através dos gráficos mais pontiagudos com quedas abruptas do número de pacotes no buffer.

Isso pode ser explicado pela diferença entre o protocolo UDP e TCP. No protocolo TCP, há contínua comunicação entre ambos os lados, tanto do que manda os pacotes segmentados quanto pelo lado que os recebe, respondendo com pacotes de reconhecimento de chegada de cada pacote. Por outro lado, pacotes UDP não possuem nenhum tipo de proteção contra corrupção ou perda de pacotes, e podem ser vistos como pacotes isolados na rede. Como o QUIC utiliza UDP, é esperado que a quantidade de pacotes na rede deva ser consideravelmente menor, e como não há resposta do servidor, quando os pacotes são enviados a fila rapidamente esvazia.

Vemos também que, apesar de sofrer do problema de *bufferbloat*, o protocolo QUIC não só é muito mais rápido mas também muito mais estável do que o protocolo TCP Reno para este caso estudado. Assim, ele deve com certeza ser considerado como um forte candidato para casos onde várias conexões devem ser feitas concorrentemente.