Avaliação de desempenho com o uso de socket TCP concorrente

Anny Caroline Correa Chagas[1]

[1]Laboratório de Ciência da Computação (LCC/IME/UERJ),
Programa de Pós-Graduação em Engenharia Eletrônica (PEL/FEN/UERJ),
Universidade do Estado do Rio de Janeiro (UERJ)

anny.chagas@uerj.br

Resumo. *Este trabalho visa analisar o desempenho de sockets TCP, com dois clientes e um servidor concorrente. Foi utilizada a linguagem Java e uma interface de 100Mbps. Cada cliente envia mensagens de tamanho 1, 2, 4, 8, ..., 32768 e 65536 bytes e o servidor responde com um echo de mesmo tamanho. Para o teste de 50.000 mensagens enviadas por cada cliente, o RTT se manteu abaixo de 1ms para mensagens de até 4096 bytes. Depois, houve um crescimento significativo, resultando em um RTT de aproximadamente 13,30ms para o tamanho máximo (65536 bytes).*

1. Introdução

Este trabalho consiste em analisar o desempenho de *sockets* TCP para diversos tamanhos de mensagens utilizando uma interface fixa de 100Mbps. Para isso, foram desenvolvidas duas aplicações utilizando a linguagem Java: a *TCPClient* e a *TCPServer*[1].

Conforme apresentado na Figura 1, as duas recebem como argumento de linha de comando a quantidade de mensagens que serão enviadas por cada tamanho de pacote (na figura, "quantidade de interações (**qIte**)"). Para **qIte** igual a 2, por exemplo, o cliente enviará duas mensagens de cada tamanho, resultando em 34 mensagens enviadas. Na prática, entretanto, será usado um número bem maior para **qIte** para se obter uma média que reflita o comportamento da rede e um intervalo de confiança de 98%.

Quando inicia, a *TCPServer* cria um *socket* passivo para aguardar conexões utilizando o número de porta também recebido como argumento de linha de comando. Após a criação do *socket*, ela passa a esperar uma primeira conexão (como queremos analisar o desempenho de dois clientes, a *TCPServer* pode esperar até duas conexões). Já a *TCPClient*, além da porta e da quantidade de iterações, recebe o endereço do *host* de destino. Nesse caso, ela deve receber o mesmo **qIte** e a mesma porta utilizados no *TCPServer*, além do endereço IP da máquina onde a aplicação *TCPServer* está executando.

Assim que começa a executar, a aplicação cliente cria duas *threads*. Cada uma representará um cliente e será responsável por enviar **qIte** mensagens por tamanho. Elas iniciam criando um *socket* e estabelecendo conexão com o servidor para depois começar o envio das mensagens seguindo um modelo "para-e-espera" (na camada de aplicação). Como *threads* executam concorrentemente, não há como determinar quem executará primeiro e nem por quanto tempo. Para a Figura 1 considerou-e que a *thread 2* (chamada de "Cliente 2") inicia executando.

[1]Todos os códigos, scripts, logs e tabelas de dados estão disponíveis em https://github.com/AnnyCaroline/sd/tree/master/l1

Quando a conexão é estabelecida, o servidor cria uma *thread* (na figura, "Servidor 1") para lidar com o recebimento das mensagens e começa a aguardar uma outra conexão, que será estabelecida quando o "Cliente 1" começar a executar.

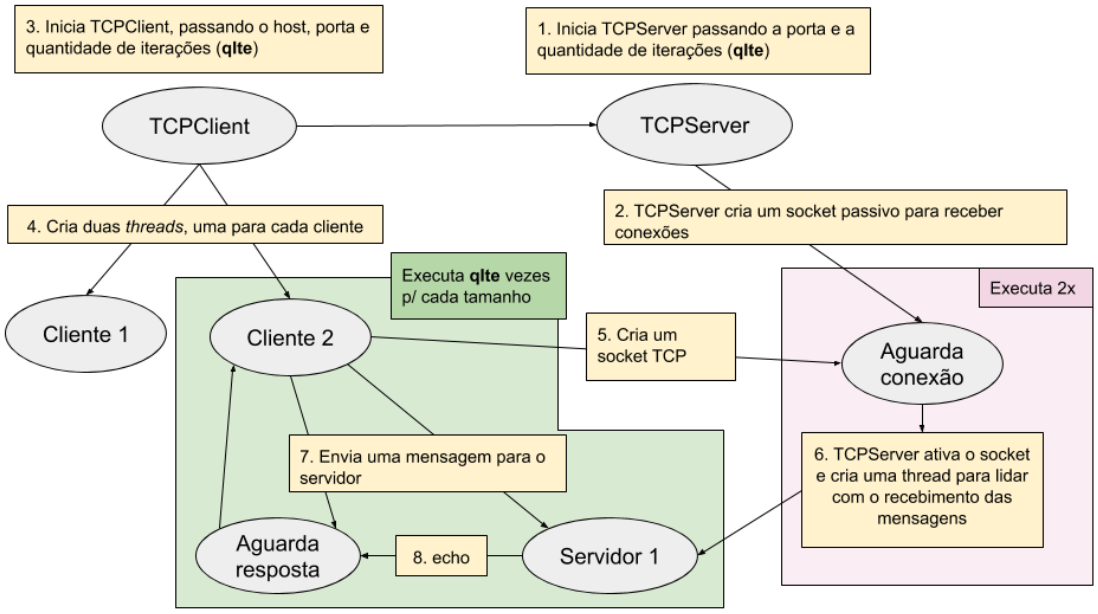

Figura 1. Visão geral das aplicações desenvolvidas

O trecho em verde na figura, representa o loop de envio e recebimento de mensagens. O Cliente 2 envia uma mensagem de tamanho 1 e passa a aguardar uma resposta do servidor, que se comporta como um *echo*, isto é, simplesmente devolve a mensagem recebida. Ao receber a resposta, o cliente pode enviar uma nova mensagem de tamanho 1, seguindo este modelo de enviar e aguardar a resposta. São enviadas **qIte** mensagens de tamanho 1 até que se comece a enviar mensagens de um tamanho maior, até chegar em 65539 bytes. Os tamanhos utilizados para as mensagens de envio e resposta foram de 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, ..., 32768, 65536 bytes (ou 2^0... 2^{16} bytes), totalizando **qIte** * 17 mensagens enviadas por cliente.

Por fim, vale citar que, como a Figura 1 não mostra a *thread* "Cliente 1" executando, a *TCPServer* só cria uma *thread*, a "Servidor 1". Com a execução da "Cliente 1", o servidor criaria uma nova *thread* (neste caso, "Servidor 2") e pararia de aguardar conexões.

A Figura 2 apresenta um exemplo de execução para o servidor: o *socket* é criado e o servidor começa a esperar uma conexão. Quando a primeira conexão é estabelecida, cria-se uma *thread* e o servidor passa a aguardar uma nova conexão, conforme foi explicado anteriormente. Diferente do caso anterior, logo em seguida a segunda conexão é estabelecida, e só depois as *threads* começam a executar.

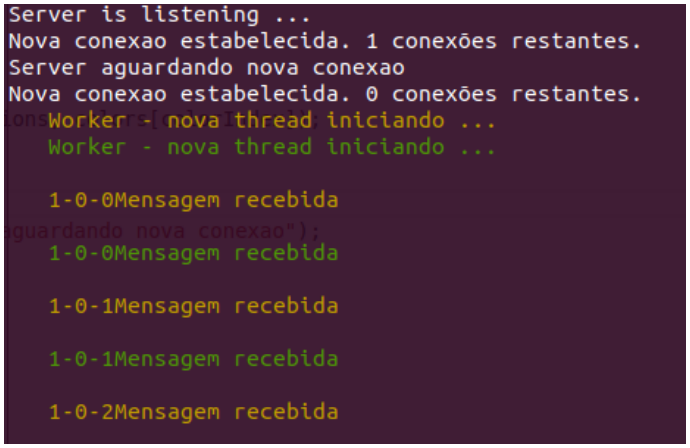

Figura 2. Servidor recebendo mensagens

2. Metodologia

Para avaliar o desempenho do uso de *sockets* TCP para diferentes tamanhos de mensagens, mediu-se o tempo médio de ida-e-volta (*round trip time* - RTT), isto é, o tempo médio para uma mensagem ser enviada do cliente para o servidor e de seu retorno do servidor para o cliente. Com essa métrica é possível analisar o atraso da rede.

Uma alternativa seria medir o tempo que uma mensagem leva do cliente para o servidor. Porém, como o sistema aqui testado é distribuído, o que significa que a *TCP-Client* e a *TCPServer* estão localizadas em máquinas diferentes, medir somente o tempo do cliente para o servidor se tornaria mais complicado, já que cada máquina possui um relógio independente. Seria necessário sincronizar os relógios das máquinas antes de se iniciarem as medições [Tanenbaum and Van Steen 2007].

A Figura 3 mostra como o sistema aqui analisado calcula o RTT. O cliente salva o momento em que uma mensagem foi enviada na variável **te** e começa a aguardar pela resposta do servidor. Esse "momento" é obtido através do método *nanoTime* da classe *System*. Ao receber o *echo* do Servidor, o cliente marca novamente o tempo, salvando-o em **tr**. Por fim, o RTT é calculado.

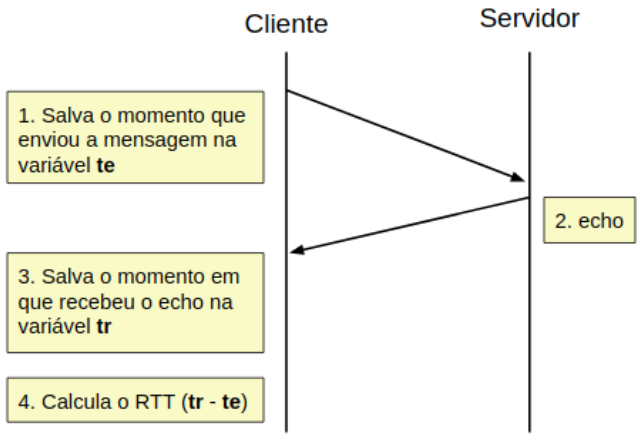

Figura 3. Cálculo do RTT

Todos os RTTs são salvos em uma lista para que, ao final de todas as iterações, seja calculada a média, o desvio padrão e o erro (com intervalo de confiança de 98%) para um dado tamanho de pacote. Esses dados podem ser logados seguindo o formato da Figura 4 ou, ainda, o formato *comma-separated values* (CSV). Este último facilita a importação dos dados por softwares como Excel e LibreOffice Calc.

```
TCP/2/100   1       2       4       8       16      32      ...
MED         0,00    0,00    0,00    0,00    0,00    0,00    ...
DVP         0,00    0,00    0,00    0,00    0,00    0,00    ...
LOW         0,00    0,00    0,00    0,00    0,00    0,00    ...
UPP         0,00    0,00    0,00    0,00    0,00    0,00    ...
```

Figura 4. Formato de exibição

3. Resultados

O primeiro teste foi realizado com 50 iterações (Figura 5). O RTT se manteve abaixo de 0,65ms para tamanhos de pacote entre 2^1 e 2^{11} bytes. O crescimento do RTT para mensagens de tamanho 2^{12} e seu decaimento para mensagens de 2^{13} pode ter sido gerado por alocações de *buffers*. Uma vez que alocação é uma tarefa custosa, é comum que se aloquem *buffers* maiores do que o necessário para evitar que haja uma realocação muito rapidamente. Portanto, o que provavelmente aconteceu foi a alocação de *buffers* maiores quando começaram a ser enviadas mensagens de tamanho 2^{12}. Para 2^{13}, não houve mais a necessidade de alocação.

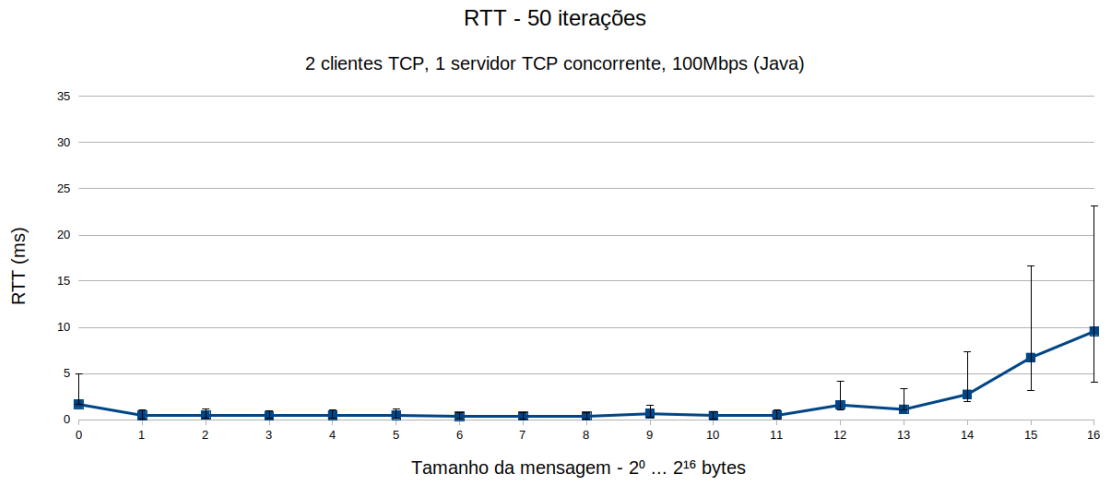

Figura 5. RTT - 50 iterações

Já o crescimento do RTT a partir de mensagens de tamanho 2^{12} pode ser explicado pela a segmentação de pacotes do protocolo TCP. O crescimento do erro e do desvio padrão a partir de pacotes desse mesmo tamanho também deve ser observado: eles indicam que, com a segmentação, o RTT se tornou mais variável e imprevisível.

Por fim, vale analisar o RTT de pacotes de 1 byte. Seu alto valor médio (1,65ms) pode ser relacionado ao *overhead* das primeiras transmissões, e também pode ser relacionado à alocação de *buffers* e ao estabelecimento da conexão. Vale observar, ainda, que a

barra de erro para pacotes de 1 byte é bem maior do que a de pacotes de 2 e 4 bytes. Isso ocorre justamente por causa do *overhead* descrito.

A Figura 6 demostra mais detalhadamente o *overhead*. Nela são apresentadas as 10 primeiras medidas de RTT de um pacote de 1 byte para a primeira conexão com o servidor. O primeiro pacote acaba tendo um RTT muito alto, enquanto os outros mantém um RTT mais próximo do esperado. Como a variação é grande, o desvio padrão e o erro também são. Para a segunda conexão (Figura 7) também há um *overhead* inicial, mas significativamente menor do que o da primeira.

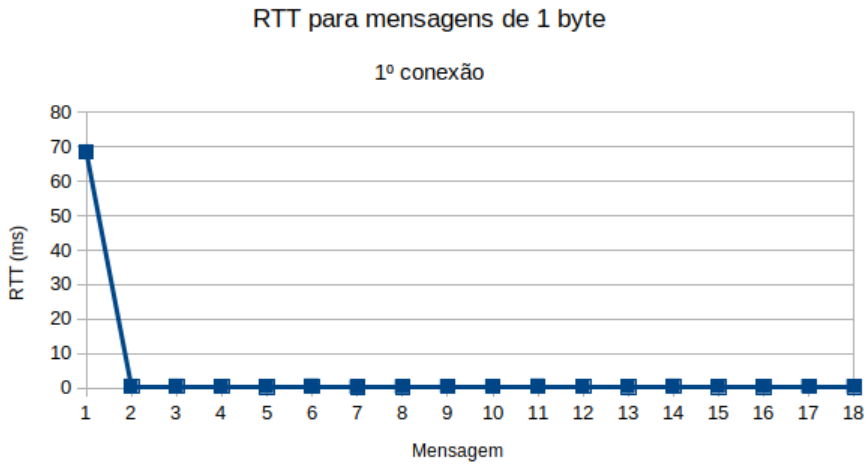

Figura 6. RTTs iniciais para mensagens de tamanho igual a 1 byte - 1º conexão

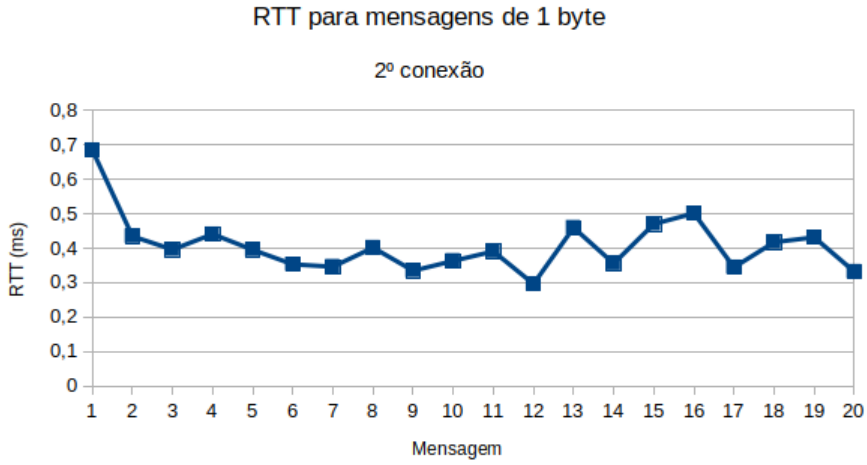

Figura 7. RTTs iniciais para mensagens de tamanho igual a 1 byte - 2º conexão

Apesar de já indicar o comportamento do sistema, 50 iterações por cliente ainda é um número muito baixo para comparar o comportamento da rede, pois deixa que *overheads* influenciem demasiadamente no resultado.

Para 100 iterações (Figura 8), já se pode observar um padrão mais próximo do esperado. O erro e a média para pacotes de tamanho igual a 1 byte diminuiu (a média

passou de 1,65 para 1,16ms). Isso porque, como estamos transmitindo mais mensagens, o *overhead* inicial se tornou menos significativo. O mesmo ocorre para pacotes de tamanho 2^{12} bytes.

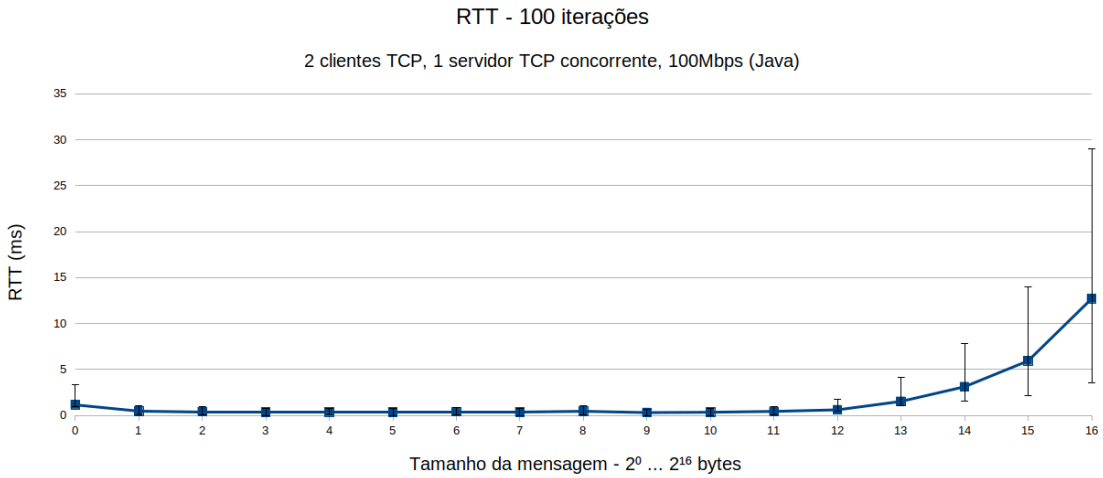

Figura 8. RTT - 100 iterações

Para 1000 iterações (Figura 9), a média para pacotes de 1 byte praticamente se igualou a de seus sucessores, e seu erro diminui significativamente. Também ficou mais claro observar que o crescimento do RTT começa a partir de pacotes de tamanho 2^{11}, e não 2^{12} como sugeria as medições anteriores. Pacotes de tamanho 2^{11} (2048 bytes) são os primeiros a serem segmentados, já que são maiores que a MTU padrão de 1500 bytes.

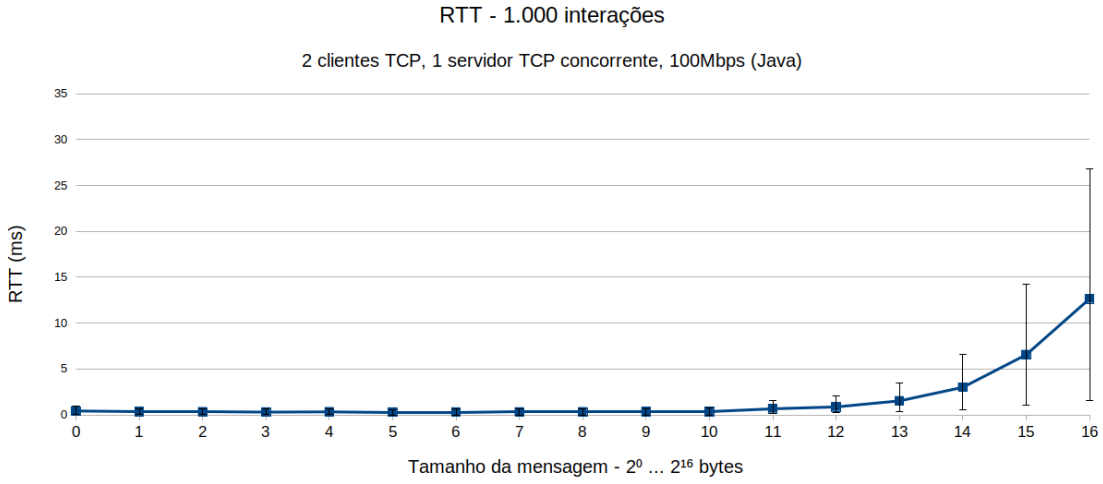

Figura 9. RTT - 1.000 iterações

Por fim, no objetivo de obter um resultado mais preciso, foi realizado um teste com 50.000 iterações. Não houveram diferenças significativas quando comparado com o teste de 1.000 iterações.

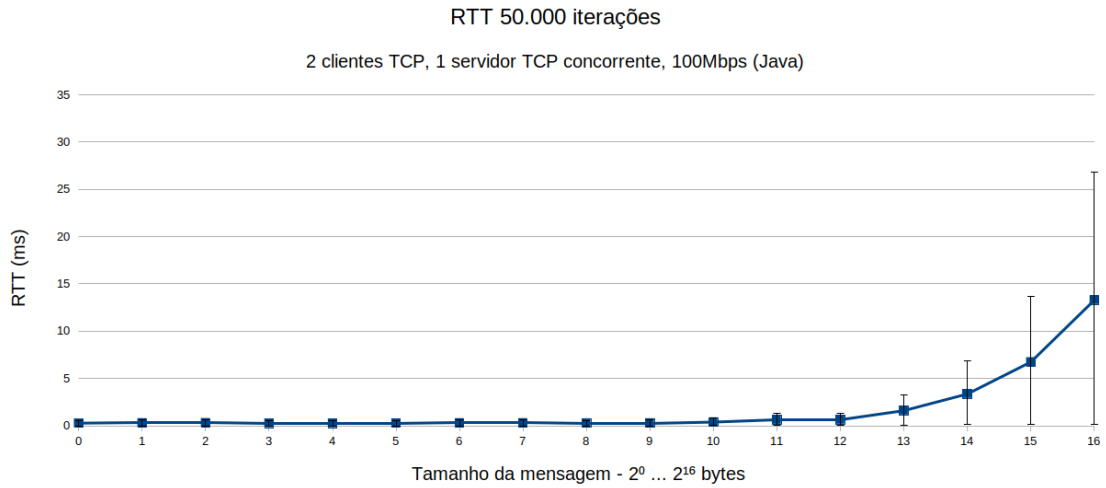

Figura 10. RTT - 50.000 iterações

4. Análise com Wireshark e tcpdump

Para melhor compreender o funcionamento de *sockets* na linguagem Java, bem como do protocolo TCP e IP, utilizamos a aplicação tcpdump e o software Wireshark[2] para monitorar os pacotes trafegados e analisá-los. Primeiramente, serão analisados os tamanhos efetivos das mensagens trafegadas, e, por último, a segmentação e a fragmentação de pacotes.

4.1. Tamanho efetivo das mensagens

para simplificar a análise e explicação, as aplicações *TCPClient* e *TCPServer* foram executadas com **qIte**=1. O tcpdump foi utilizado para monitorar as mensagens enviadas e recebidas pelo cliente. O resultado do monitoramento foi salvo em um arquivo para posterior análise utilizando o Wireshark. A Figura 11 apresenta alguns pacotes que trafegaram de/para a aplicação *TCPClient*. Para simplificar, foram listados somente os pacotes que contêm dados, utilizando, para isso, o filtro **tcp.len>0** no Wireshark. Foram omitidos da Figura 11, portanto, pacotes ACK e de sincronização e finalização, utilizados, respectivamente, para o *three-way handshake*[3] do protocolo TCP e para a finalização da conexão.

Voltando à Figura 11, é possível verificar que dois pacotes com tamanhos iguais são transmitidos da *TCPClient* (10.0.100.11) para a *TCPServer* (10.0.100.16), um por cada *thread*. Da mesma forma, há duas respostas de igual tamanho (vide coluna *Length*).

O tamanho da mensagem transmitida foi efetivamente a quantidade desejada. Houve o *overhead* somente dos cabeçalhos de outros protocolos (32 bytes para o TCP[4], 20 bytes, IP e 14 bytes, Ethernet), o que é o esperado em uma rede de comutação de pacotes. A Figura 12 apresenta uma análise detalhada de um pacote, que possui 1 byte de

[2]https://www.wireshark.org/

[3]Na verdade, o terceiro segmento enviado devido ao *three-way handshake* pode carregar uma carga útil [Kurose 2013]. Porém, isso não foi observado neste trabalho.

[4]Um cabeçalho TCP normalmente possui 20 bytes. Porém, os *sockets* oferecidos pela linguagem Java usam 12 bytes de opções.

No.	Time	Source	Destination	Protoco	Lengt
7	0.003075	10.0.100.11	10.0.100.16	TCP	67
8	0.003090	10.0.100.11	10.0.100.16	TCP	67
11	0.073111	10.0.100.16	10.0.100.11	TCP	67
13	0.073360	10.0.100.16	10.0.100.11	TCP	67
15	0.073490	10.0.100.11	10.0.100.16	TCP	68
16	0.073614	10.0.100.11	10.0.100.16	TCP	68
19	0.074369	10.0.100.16	10.0.100.11	TCP	68
21	0.074527	10.0.100.16	10.0.100.11	TCP	68
20	0.074483	10.0.100.11	10.0.100.16	TCP	70
22	0.074585	10.0.100.11	10.0.100.16	TCP	70
23	0.074878	10.0.100.16	10.0.100.11	TCP	70
24	0.074929	10.0.100.16	10.0.100.11	TCP	70

Figura 11. Alguns pacotes transmitidos e recebidos pela *TCPClient*

dados (última linha) e um tamanho total de 67 bytes (contando os cabeçalhos - primeira linha).

```
▸ Frame 7: 67 bytes on wire (536 bits), 67 bytes captured (536 bits)
▸ Ethernet II, Src: 52:5a:89:5e:87:0c (52:5a:89:5e:87:0c), Dst: 92:50:c3:9f:14:c7 (92:50:c3:9f:14:c7)
▸ Internet Protocol Version 4, Src: 10.0.100.11, Dst: 10.0.100.16
▸ Transmission Control Protocol, Src Port: 32778, Dst Port: 1025, Seq: 1, Ack: 1, Len: 1
▸ Data (1 byte)
```

Figura 12. Análise de um pacote com 1 byte de dados

4.2. Segmentação e TCP Segmentation Offload

Para mensagens com tamanho até 1024 bytes, foi possível observar com clareza a quantidade de dados transmitida. Para mensagens maiores, essa visualização não foi tão clara, já que os pacotes precisam ser segmentados e/ou fragmentados para respeitar a unidade máxima de transmissão (*maximum transmission unit* - MTU) padrão de 1500 bytes.

A quantidade máxima de dados que podem ser armazenados em um segmento TCP é definida pelo MSS (*maximum segment size*). Em geral, o MSS usa como base a MTU do enlace de saída, de forma a garantir que o segmento TCP criado possa ser encapsulado em um datagrama IP sem ultrapassar a MTU. O MSS também pode ser definido com base no menor MTU do caminho — o maior quadro de camada de enlace que pode ser enviado por todos os enlaces desde a origem até o destino [Kurose 2013]. Como, para esse trabalho, o *host* cliente e o servidor estão ligados diretamente, a menor MTU do caminho é a do enlace de saída do cliente: 1500 bytes.

Com base nisso, o esperado é que a medição realizada com o tcpdump mostre a segmentação, e que possamos calcular a quantidade de dados transmitidos observando o *overhead* de cabeçalhos e a quantidade de segmentos transmitidos. Porém, como mostra a Figura 13, essa visualização não é clara. Inclusive, há a transmissão de pacotes muito maiores do que a MTU. Como isso é possível?

Normalmente a segmentação é realizada pela CPU hospedeira, que entrega ao adaptador de rede (*Network Interface Card* - NIC) os dados já segmentados respeitado o MSS e, por consequência, a MTU. Para otimizar o uso de recursos do *host*, é possível delegar a segmentação para a NIC (utilizando um processo chamado *TCP segmentation offload*. Dessa forma, a CPU hospedeira passa a enviar pedaços de dados para a NIC, ao invés dos dados já segmentados [TribeLab 2016].

No.	Time	Source	Destination	Protoco	Length	Info
134	0.088122	10.0.100.16	10.0.100.11	TCP	2962	1025 → 32778 [PSH, ACK] Seq=
135	0.088135	10.0.100.11	10.0.100.16	TCP	66	32778 → 1025 [ACK] Seq=32768
1…	0.088164	10.0.100.16	10.0.100.11	TCP	14546	1025 → 32778 [ACK] Seq=33760
137	0.088254	10.0.100.11	10.0.100.16	TCP	15994	32778 → 1025 [ACK] Seq=32768
138	0.088645	10.0.100.16	10.0.100.11	TCP	15994	1025 → 32778 [ACK] Seq=48240
139	0.088687	10.0.100.11	10.0.100.16	TCP	15994	32778 → 1025 [ACK] Seq=48696
140	0.088700	10.0.100.11	10.0.100.16	TCP	978	32778 → 1025 [PSH, ACK] Seq=
141	0.088706	10.0.100.16	10.0.100.11	TCP	1434	1025 → 32778 [PSH, ACK] Seq=
142	0.088938	10.0.100.16	10.0.100.11	TCP	66	1025 → 32778 [ACK] Seq=65536
143	0.092055	10.0.100.11	10.0.100.16	TCP	17442	32778 → 1025 [ACK] Seq=65536
144	0.093676	10.0.100.16	10.0.100.11	TCP	14546	1025 → 32778 [ACK] Seq=65536
145	0.093775	10.0.100.11	10.0.100.16	TCP	20338	32778 → 1025 [ACK] Seq=82912
146	0.093810	10.0.100.16	10.0.100.11	TCP	7306	1025 → 32778 [ACK] Seq=80016
147	0.094217	10.0.100.11	10.0.100.16	TCP	15994	32780 → 1025 [ACK] Seq=32768
148	0.094589	10.0.100.16	10.0.100.11	TCP	14546	1025 → 32778 [PSH, ACK] Seq=

Figura 13. Análise com gso e tso ativados

Apesar de otimizar os recursos do hospedeiro, o *TCP segmentation offload* acaba dificultando a monitoração. Isso porque o tcpdump monitora justamente os dados que saem da CPU e não da NIC (Figura 14).

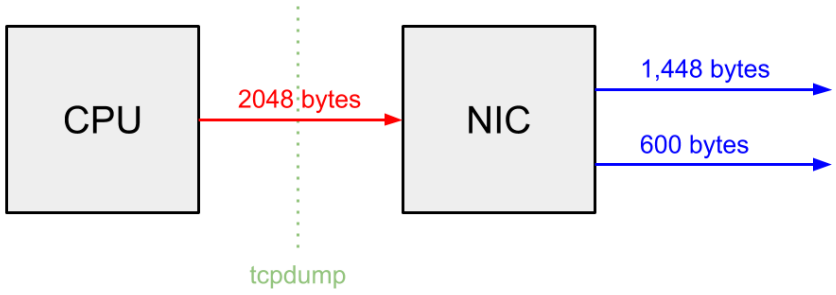

Figura 14. TCP segmentation offload

Portanto, para se prosseguir com os testes deste trabalho, foi necessário desabilitar as opções **generic-segmentation-offload** (gso) e **tcp-segmentation-offload** (tso) dos *hosts* [Rtoodtoo 2011].

4.3. Análise da segmentação

Com as opções gso e tso desabilitadas, as aplicações foram executadas novamente. A Figura 15 mostra os pacotes de maior tamanho. Veja que agora eles não ultrapassam 1514 bytes. Considerando que o cabeçalho da camada de enlace possui 14 bytes, e que a MTU se refere ao datagrama IP, está conforme esperávamos. Vale notar, também, o tamanho dos dados efetivamente transmitidos, isto é, sem considerar os cabeçalhos: 1448 bytes.

Para um melhor entendimento, vamos analisar um último exemplo. Foram criadas

No.	Time	Source	Destination	Protoco	Lengt	Info
430	0.133217	10.0.100.16	10.0.100.11	TCP	1514	1025 → 33450 [ACK] Seq=128336 Ack=131072 Win=14944 Len=1448
429	0.133212	10.0.100.16	10.0.100.11	TCP	1514	1025 → 33450 [ACK] Seq=126888 Ack=131072 Win=14944 Len=1448
428	0.133208	10.0.100.16	10.0.100.11	TCP	1514	1025 → 33450 [ACK] Seq=125440 Ack=131072 Win=14944 Len=1448
427	0.133202	10.0.100.16	10.0.100.11	TCP	1514	1025 → 33450 [PSH, ACK] Seq=123992 Ack=131072 Win=14944 Len=
426	0.133170	10.0.100.16	10.0.100.11	TCP	1514	1025 → 33450 [ACK] Seq=122544 Ack=131072 Win=14944 Len=1448
425	0.133166	10.0.100.16	10.0.100.11	TCP	1514	1025 → 33450 [PSH, ACK] Seq=121096 Ack=131072 Win=14944 Len=
424	0.133162	10.0.100.16	10.0.100.11	TCP	1514	1025 → 33450 [PSH, ACK] Seq=119648 Ack=131072 Win=14944 Len=
423	0.133158	10.0.100.16	10.0.100.11	TCP	1514	1025 → 33450 [ACK] Seq=118200 Ack=131072 Win=14944 Len=1448

Figura 15. Análise com gso e tso desativados

duas aplicações de teste, uma para o cliente e outra para o servidor. O cliente simplesmente envia uma mensagem de tamanho igual a 2048 bytes para o servidor através de um *socket* TCP. O servidor é encerrado logo após o recebimento da mensagem.

A Figura 16 apresenta um gráfico de fluxo gerado pelo próprio Wireshark. O cliente, localizado no *host* 10.0.100.11 e com a porta 33348, inicia o *three-way handshake* enviando uma mensagem sem carga útil, mas com a *flag* SYN habilitada. O número de sequencia inicia como 0, já que é o primeiro pacote que será transmitido. Por padrão, o ACK também incia com 0, apesar de não ser mostrado na figura.

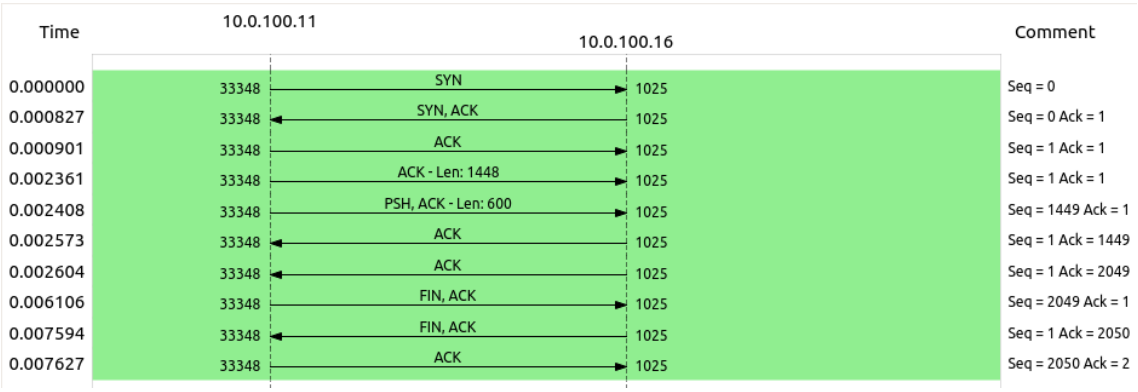

Figura 16. Gráfico de fluxo com segmentação de um pacote

A resposta do servidor (localizado na máquina 10.0.100.16, na porta 1025) também não carrega carga útil, mas possui as *flags* SYN e ACK habilitadas. Isso significa que a mensagem foi corretamente recebida (ACK) e que o servidor também deseja estabelecer uma conexão (SYN). O número de sequência também é 0, pois é a primeira mensagem que o servidor está transmitindo. O ACK, por outro lado, é igual a 1. Na fase de conexão, o ACK não indica a quantidade de bytes de carga útil recebidas. Ele simplesmente é setado como 1 para indicar que um pacote com a *flag* SYN foi recebido. Para finalizar o *three-way handshake*, o cliente responde o número de sequência 0 do servidor com um ACK igual a 1. Seu número de sequência também é setado como 1 [Packetlife 2010].

Com a conexão estabelecida, o cliente pode começar a enviar os dados. Como citado no começo desta seção, o cliente necessita enviar 2048 bytes de dados. Encapsulando essa carga útil em um cabeçalho TCP/IP temos como total 2100 bytes (32 do TCP e 20 do IP), mais do que do que é possível ser enviado. Portanto, é necessário que haja segmentação.

O primeiro segmento possui 1448 bytes de carga útil, mais 52 bytes de cabeçalho TCP/IP, totalizando 1500 bytes, o esperado já que estamos usando a MTU padrão. O

ACK também é enviado junto, e continua sendo igual a 1. Logo em seguida, o cliente envia um pacote de carga útil igual a 600, completando os 2048 bytes que precisavam ser enviados. É importante notar que não foi necessário esperar o recebimento do ACK do servidor para o envio de um segundo segmento, já que o TCP não é um protocolo para-e-espera. O servidor responde com dois ACKs, informando, respectivamente, que já foram recebidos 1449 e 2049 bytes.

As três últimas mensagens são relativas ao fechamento da conexão, que funciona de maneria similar ao *three-way handshake*. O cliente envia uma mensagem sem carga útil e com a *flag* FIN habilitada. O servidor responde com uma mensagem também sem carga útil e com a *flag* e, por fim, o cliente envia um ACK para finalizar o fechamento.

Analisando em mais detalhes um segmento TCP com carga útil, percebe-se mais um ponto interessante. O cabeçalho IP apresenta a *flag Don't fragment* (DF). Isso significa que se um pacote IP chegar a um roteador que possua uma MTU menor do que seu tamanho, ele não poderá ser fragmentado e será descartado.

Essa *flag* é geralmente setada em pacotes IP que transportam segmentos TCP para dar mais flexibilidade ao TCP ajustar seu MSS dinamicamente. Com o sinalizador DF definido, um pacote ICMP deve ser retornado se um roteador intermediário tiver que descartar um pacote porque é muito grande. O TCP remetente irá então reduzir sua estimativa da MTU do caminho da conexão e reenviar os pacotes necessários em segmentos menores. Se o DF não fosse definido, o TCP de envio nunca saberia que estava enviando segmentos muito grandes. Esse processo é chamado PMTU-D (*Path MTU Discovery*) [Stackoverflow 2010].

```
▸ Frame 4: 1514 bytes on wire (12112 bits), 1514 bytes captured (12112 bits)
▸ Ethernet II, Src: 52:5a:89:5e:87:0c (52:5a:89:5e:87:0c), Dst: 92:50:c3:9f:14:c7 (92:50:c3:9f:14:c7)
▾ Internet Protocol Version 4, Src: 10.0.100.11, Dst: 10.0.100.16
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  ▸ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 1500
    Identification: 0x0365 (869)
  ▾ Flags: 0x4000, Don't fragment
      0... .... .... .... = Reserved bit: Not set
      .1.. .... .... .... = Don't fragment: Set
      ..0. .... .... .... = More fragments: Not set
      ...0 0000 0000 0000 = Fragment offset: 0
    Time to live: 64
    Protocol: TCP (6)
    Header checksum: 0x559c [validation disabled]
    [Header checksum status: Unverified]
    Source: 10.0.100.11
    Destination: 10.0.100.16
▸ Transmission Control Protocol, Src Port: 33348, Dst Port: 1025, Seq: 1, Ack: 1, Len: 1448
▸ Data (1448 bytes)
```

Figura 17. Detalhes de um pacote com carga útil igual a 1448 bytes

5. Análise da capacidade

A capacidade da rede foi analisada por meio do aumento do número de clientes, de forma a aumentar a taxa de requisições em uma faixa de tempo.

Conforme mostra a Figura 18, mensagens pequenas (até 2^3 bytes) não parecem ser prejudicadas pelo aumento de clientes. Já mensagens de 2^{16} bytes saltam de menos de 15ms para aproximadamente 48ms com apenas 8 clientes.

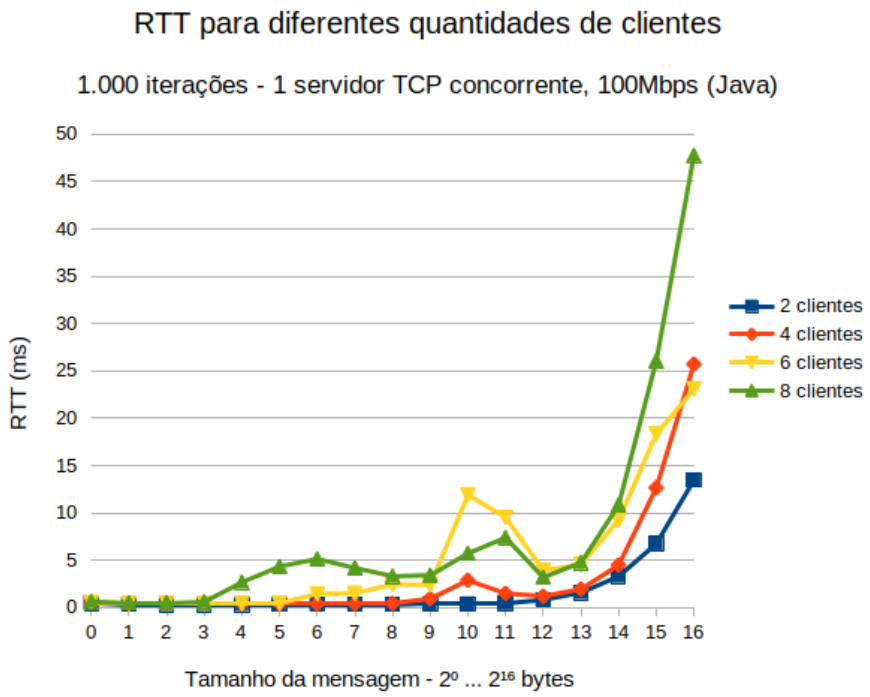

Figura 18. RTT para diferentes quantidades de clientes

Referências

Kurose, J. F. (2013). *Redes de computadores e a internet - uma abordagem top-down*. Pearson Education do Brasil, 6 edition.

Packetlife (2010). Understanding tcp sequence and acknowledgment numbers.

Rtoodtoo (2011). Generic/tcp segmentation offload and wireshark.

Stackoverflow (2010). Benefits of “don’t fragment” on tcp packets.

Tanenbaum, A. S. and Van Steen, M. (2007). *Distributed systems: principles and paradigms*. Prentice-Hall.

TribeLab (2016). Analyzing tcp segmentation offload with wireshark.