Trabalho Prático 1 - Redes de Computadores
Relatório de Desempenho

Alexander Decker de Sousa[1], **Eduardo Pinto Mendes Câmara Júnior**[1]

[1]Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG)

{epmcj,alexanderdecker}@dcc.ufmg.br

1. Introdução

A Internet proporcionou muitas facilidades para o nosso dia a dia. Uma delas é a possibilidade de realizar a transferências de arquivos entre dois computadores através da rede. Esse trabalho tem como objetivo desenvolver aplicações para esta finalidade.

As duas aplicações desenvolvidas seguem o modelo cliente-servidor, no modo requisição-resposta sobre o protocolo TCP e a versão 6 do IP (IPv6). O cliente é capaz de solicitar ao servidor uma lista dos arquivos disponíveis para *download* e baixar um deles. Para isso, o cliente conta com dois comandos: **list** e **get**. O comando **list** solicita ao servidor a lista de arquivos disponíveis para *download* e os exibe na tela. Já o comando **get** solicita um arquivo ao servidor e o recebe caso ele esteja disponível.

A aplicação do cliente possui o nome **clienteFTP** e seus comandos seguem as formas mostradas abaixo. Ela também é capaz de medir a vazão alcançada durante uma transferência de arquivo.

- clienteFTP list <nome ou IPv6 do servidor><porta do servidor><tamanho do *buffer*>
- clienteFTP get <nome do arquivo><nome ou IPv6 do servidor><porta do servidor><tamanho do *buffer*>

O programa do servidor é chamado **servidorFTP** e é capaz de processar múltiplas conexões simultâneas. Ele é ativado através do seguinte comando:

- servidorFTP <porta do servidor><tamanho do *buffer*><diretório a ser utilizado>

Em ambos programas, o campo <tamanho do *buffer*>representa o tamanho do *buffer* para envio/recebimento de dados.

Vale notar que é possível ocorrer erros em ambas as aplicações. Exemplos de erros podem ser encontrados quando um dos lados fecha a conexão antes da hora prevista ou quando um cliente solicita um arquivo que não existe no servidor. Quando estes e outros erros acontecem, uma mensagem deve ser exibida. Tal mensagem deve informar o código e a descrição do erro, conforme especificado na Tabela 1.

Dito tudo isso, o restante do trabalho apresenta questões chaves de implementação do projeto e detalhes de desempenho. A seção 2 discute as decisões de implementação adotadas durante o desenvolvimento das aplicações. A seção 3 apresenta informações sobre os experimentos realizados. Já a seção 4 apresenta as informações coletadas durante a realização dos experimentos e a seção 5 discute os resultados observados. Por fim, são feitas as considerações finais na seção 6.

Tabela 1. Códigos e Descrições dos Erros

Código de erro	Descrição do erro
-1	Erro nos argumentos de entrada
-2	Erro na criação do *socket*
-3	Erro de *bind*
-4	Erro de *listen*
-5	Erro de *accept*
-6	Erro de *connect*
-7	Erro de comunicação com cliente/servidor
-8	Arquivo solicitado não encontrado
-9	Erro de ponteiro
-10	Comando de clienteFTP não existente
-999	Outros erros

2. Decisões de Implementação

Durante o desenvolvimento das aplicações, algumas decisões precisaram ser tomadas. A primeira, e mais importante, é como seria o formato das mensagens trocadas entre clientes e o servidor. Visando um menor impacto na vazão de dados, optamos por:

- O primeiro byte das mensagens enviadas do cliente para o servidor sinalizam qual o comando pretendido. Assim, a mensagem para o comando **list** possui somente um byte com o valor igual a 'L'. Já a mensagem do comando **get** possui o primeiro byte com o valor 'G' e os bytes seguintes contém o nome do arquivo a ser transferido.
- As mensagens de erro transferidas do servidor para o cliente sempre são compostas por 2 bytes: o primeiro com o valor 'E' e o outro com o valor absoluto do código de erro (com exceção do código -999, que é representado pelo valor 11 devido ao problema de *overflow*).
- As mensagens de resposta do servidor possuem estruturas diferentes.
  - As mensagens de resposta ao comando **list** seguem o seguinte padrão: O primeiro byte da primeira mensagem indica se a mensagem é de erro (seguindo o devido formato das mensagens de erro) ou se é de fato a lista de arquivos (com o carácter 'O'). Neste último caso, as mensagens seguintes contém somente dados sobre a lista e os envios terminam após o envio do '\0'.
  - As mensagens de resposta ao comando **get** sempre possuem um byte (o primeiro) que identifica o conteúdo da mensagem. O valor dele pode ser 'E', para sinalizar erro, ou então 'O' ou 'F' para dizer que o conteúdo é de dados do arquivo. A diferença entre os dois últimos valores é que o 'F' é utilizado para sinalizar a última mensagem a ser enviada, enquanto o 'O' é utilizado no restante delas.

A segunda decisão foi com respeito ao fornecimento de acessos múltiplos. Quando o usuário manda um comando para o servidor, é alocada um *thread* para tratar a requisição e depois é utilizada a função *pthread_create()* para dispará-la. Optamos por não limitar o número máximo de *threads* que podem ser disparadas durante a execução do servidor, ficando assim limitado pelo tamanho da memória.

Optou-se por não utilizar a lista encadeada para implementar a resposta ao comando **list**. Ao invés disso, o servidor já envia a lista de forma textual ao cliente, que por sua vez apenas a imprime na tela. Essa mudança foi considerada para simplificar a implementação e evitar eventuais erros.

A última decisão tem relação com as mensagens de erros. O código -1 é retornado sempre que existe um erro nos argumentos de entrada do programas. Ou seja, ele é mostrado se existem parâmetros faltando, passando ou se o diretório especificado no servidor não existe. Os outros erros possuem lugares bem específicos, com exceção do -999. Optamos por utilizar esse código somente no caso em que é requisitada a lista de arquivos de um diretório vazio. É importante notar que nenhum código adicional de erro foi acrescentado.

3. Metodologia

Foram realizados três experimentos de forma a avaliar a aplicação em termos da taxa de transferência (vazão) de arquivos. Tal taxa consiste na razão entre a quantidade de bytes enviada ou recebida pelo cliente e o intervalo de tempo desde o término na avaliação dos argumentos da linha de comando até o fechamento do *socket*, ou seja, da última operação significativa da execução. Em todos os casos, escolheu-se um tamanho fixo para o arquivo a ser transferido e variou-se o tamanho do *buffer*, utilizando uma escala logarítmica em potências de dois. Para cada tamanho de *buffer*, foram realizadas três medições, de forma a considerar a média destas. Foi calculado também o desvio padrão de cada trio de medições, de forma a permitir a análise do bom comportamento da curva e compor a barra de erros nos pontos dos gráficos.

Optou-se por gerar os dados executando cada programa múltiplas vezes, de forma a avaliar de fato o código que foi enviado, ao invés de avaliar uma adaptação deste código que automatizasse os testes.

Três experimentos foram realizados localmente, tendo a aplicação cliente e a servidor rodando na mesma máquina. É esperado que estas sejam bem comportadas, ao contrário de medições entre máquinas, já que são menos influenciadas por fatores do ambiente. É esperado também que a fração da latência efetivamente gasta com a transferência de bytes seja muito pequena, já que as mensagens serão trocadas apenas a nível de processos da mesma máquina. Por fim, a parcela da latência gasta com processamento local deve também ser significativa, de forma que o aumento do tamanho do buffer não seja tão impactante quanto seria em uma transferência entre *hosts* distintos.

Dois experimentos foram realizados entre as máquinas *cipo.grad.dcc.ufmg.br* (servidor) e *eufrates.grad.dcc.ufmg.br* (cliente). Como se trata de dois *hosts* distintos, é esperado um comportamento mais caótico do que nos testes anteriores, devido à menor confiabilidade do canal a nível físico e susceptividade a colisões e ruídos, além de outras limitações diversas. A parcela da latência determinada pela transferência de bytes propriamente dita deve também ser maior do que nos casos anteriores, sendo a parcela causada por processamento local possivelmente insignificante.

Nos dois casos, é esperado que a vazão cresça com o aumento do *buffer*, já que os arquivos poderão ser transferidos em menos mensagens. A vazão deve se estabilizar à medida em que o tamanho do buffer se aproxima do tamanho do arquivo, e então decair,

visto que será transferida uma mensagem maior do que o arquivo em si. Não obstante, no nosso caso, esse decaimento não será observado, visto que a composição das mensagens não admite subutilização.

4. Resultados

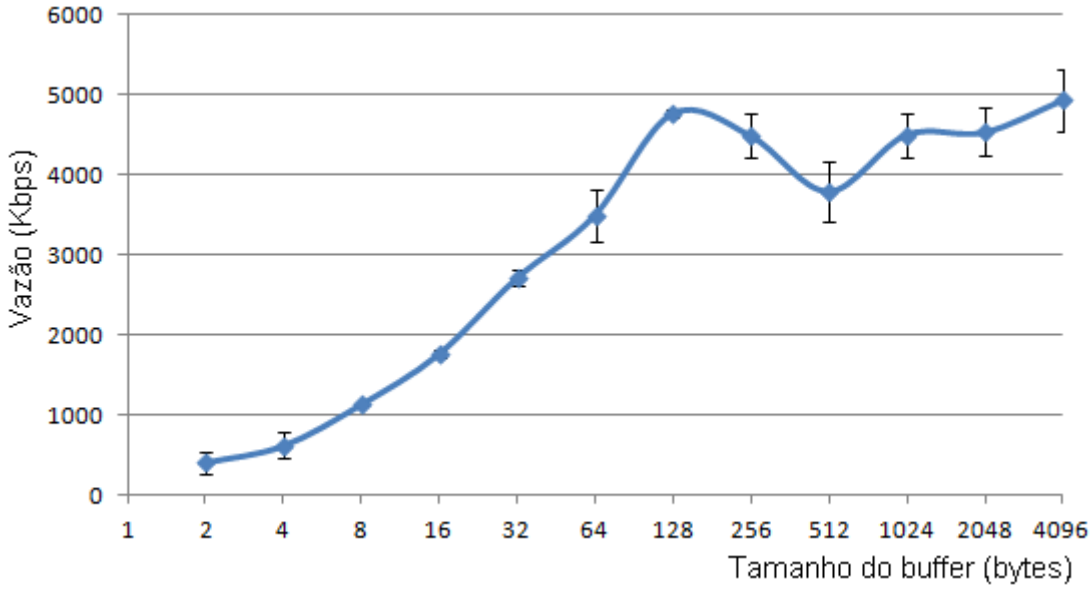

Figura 1. Vazão da transferência de um arquivo de 2,3 KB em relação ao tamanho do *buffer*, dentro do *Localhost*.

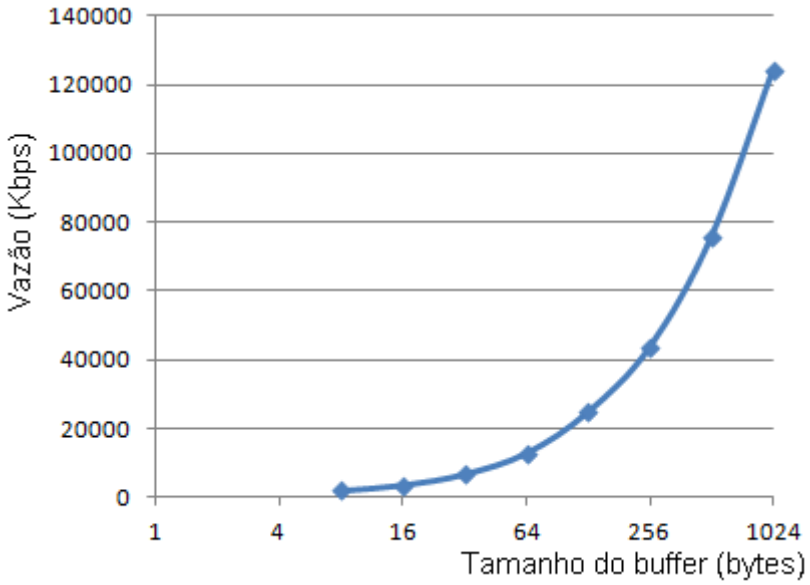

Figura 2. Vazão da transferência de um arquivo de 4,5 MB em relação ao tamanho do *buffer*, dentro do *Localhost*. O intervalo do eixo X foi escolhido para facilitar a visualização do ganho de vazão em função do aumento do *buffer*, nos casos em que o tamanho do *buffer* ainda é muito menor do que o tamanho do arquivo.

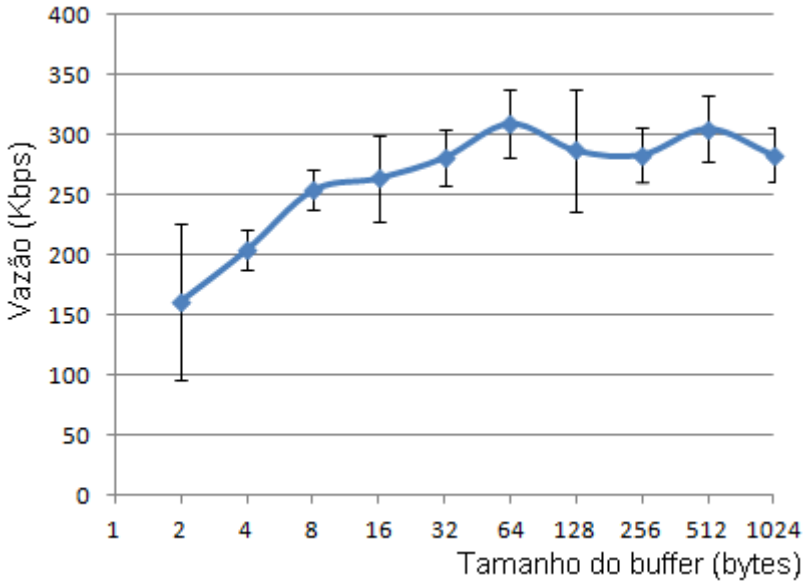

Figura 3. O gráfico mostra a vazão da transferência de um arquivo muito pequeno (220B) dentro do *Localhost*, de acordo com diversos tamanhos de *buffer*

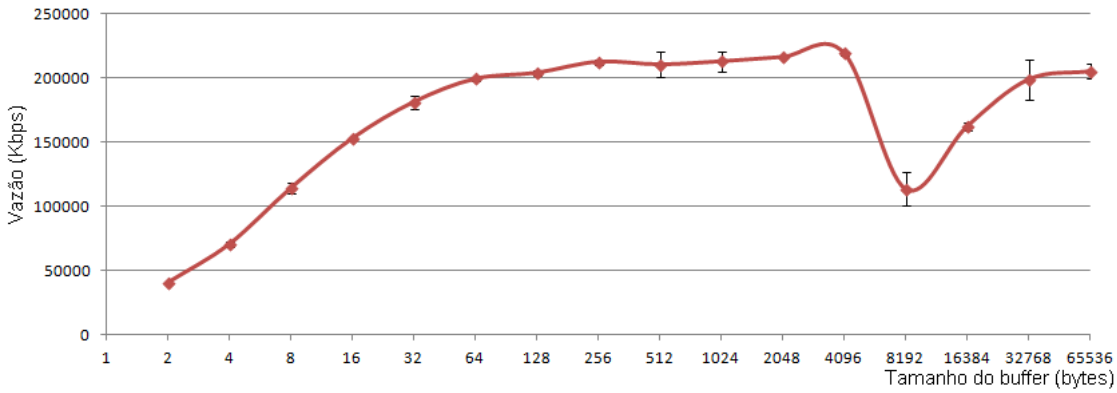

Figura 4. Vazão da transferência de um arquivo de 3,5 MB em relação ao tamanho do *buffer*. Transferência realizadas entre *hosts* diferentes.

5. Análise de Dados

De fato a vazão aumenta exponencialmente com o aumento do tamanho da mensagem até o momento em que o tamanho do arquivo é atingido, como se pode ver nas Figuras 1, 4 e 3. Após isso, a vazão se estabiliza, como pode ser observado nas Figuras 1 e 3.

Em geral não há decaimento da vazão com o aumento excessivo, apenas em casos em que o fator caótico ambiental é alto, como na Figura 4.

É possível observar que a influência do tamanho do *buffer* em arquivos pequenos é muito baixa, de forma que a vazão se estabiliza muito rápido, como vemos nas Figuras 3 e 5. No caso da Figura 5, erros e atrasos são extremamente evidenciados, pelo fato de poucas mensagens serem trocadas e da transferência ser entre *hosts* diferentes e, portanto, sob alta influência do ambiente.

Tabela 2. Representação em tabela do experimento exposto na Figura 4

Tamanho da mensagem	Bytes transferidos	Tempo (s)	Vazão Média (kbps)
2	24517664	4,845613333	40478,33333
4	16345113	1,855461333	70501,42667
8	14010098	0,982573667	114158,88
16	13076093	0,684224	152886,7767
32	12654284	0,558876667	181227,84
64	12453422	0,499927333	199291,4533
128	12355364	0,484676333	203953,96
256	12306911	0,463124667	212603,5667
512	12282827	0,467524333	210485,57
1024	12270821	0,461529667	212880,94
2048	12264826	0,453661	216283,8533
4096	12261831	0,447333667	219294,3367
8192	211951	0,015108	113302,3167
16384	941039	0,04645	162114,52
32768	2497519	0,100979667	198769,96
65536	3677167	0,143470333	205151,1033

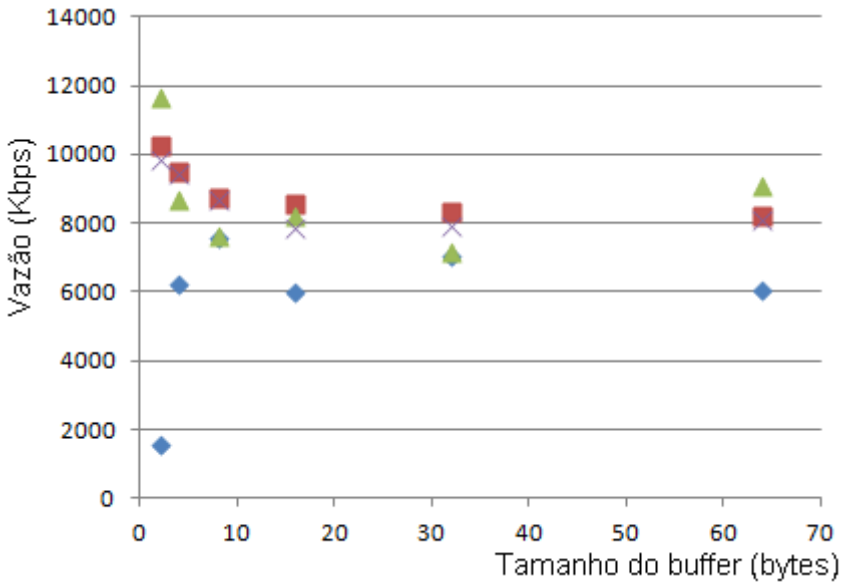

Figura 5. Transferência de um arquivo pequeno (4KB) entre *hosts* distintos. Foram realizadas quatro séries de medições. Cada série foi representada com um padrão visual diferente a fim de ser possível ver a natureza caótica do conjunto. A média das vazões permanece mais ou menos constante com o aumento do tamanho do *buffer*, enquanto seu desvio padrão cai bastante.

6. Conclusão

O processo de desenvolvimento de aplicações que seguem o modelo de operação cliente-servidor se mostrou uma experiência bastante positiva. Foi possível exercitar a utilização de *sockets* através das operações de *listen*, *connect*, *accept*, *send* e *receive*. Também foi possível verificar desafios em se realizar a comunicação entre dois computadores. Foi

preciso especificar bem os formatos das mensagens trocadas entre os clientes e o servidor pra que não ocorresse nenhum problema.

Depois de desenvolver as aplicações, foi preciso testá-las. Na realização dos experimentos, pudemos verificar a relação da vazão em função do tamanho da mensagem. A vazão cresceu com o aumento da tamanho da mensagem até que esta atingisse ou ultrapassasse o tamanho do arquivo que seria transmitido. Também pudemos verificar a influência do meio na transmissão de mensagens entre dois *hosts* diferentes. Algumas vezes, a vazão de dados foi menor do que a prevista provavelmente devido ao uso do canal por mais transmissões.