Projeto 2 - Cliente e Servidor - UDP e TCP

Gabriel Affonso - 167710 e Luiz Eduardo Cartolano - 183012
MC833 - Turma A

Maio de 2019

Resumo

Análise de tempo de comunicação da camada de transporte em protocolos TCP e UDP no contexto de um sistema de banco de dados de perfis pessoais com objetivo de comparar eficiência e confiabilidade com outros protocolos. Ambos mostraram-se bastante confiáveis, pelo menos em testes na rede interna do Instituto de Computação Unicamp. Testes externos foram impossíveis devido a falta de acesso aos roteadores da equipe para realizar port-forwarding. O tempo de transporte foi menor para o protocolo UDP, como esperado na literatura.

1 Introdução

O modelo de arquitetura de computadores *cliente-servidor* é um dos mais populares no mundo. Usado em praticamente todos os sistemas automatizados de bibliotecas, ele separa a aplicação em duas partes distintas. O *cliente* - responsável por fazer requisições - e o *servidor* - responsável por prover as informações requisitadas. Uma exemplificação do modelo de funcionamento pode ser vista na Figura 1.

A comunicação entre ambas as partes usualmente é regida por protocolos padrões que são mundialmente conhecidos, como *UDP* e *TCP*. Os protocolos de transporte fornecem uma interface para as camadas superiores de abstração trocarem dados sem se preocupar com as complexidades inerentes ao transporte físico dos dados, que é *unreliable*. Assim, é garantido que as aplicações seguirão padrões mínimos de qualidade.

Para este trabalho serão usados ambos os protocolos, e seus resultados serão comparados, a fim de se entender as diferenças entre eles, bem como as vantagens e desvantagens de seus usos.

2 Sistema

2.1 Descrição Geral

O modelo de arquitetura *cliente-servidor* descreve como um servidor provê recursos e serviços para um ou mais clientes. Este arquétipo pode ser implementado em uma miríade de protocolos. Os dois escolhidos para serem implementados e comparados foram o *TCP* Permanente e *UDP* por diferirem no seu *overhead*.

2.2 Transmission Control Protocol (TCP) Permanente

O *TCP* permanente é "orientado à conexão", significando que o cliente estabelece uma conexão com o servidor e ambos trocam múltiplas mensagens de tamanhos variados, sendo a aplicação do cliente quem termina a sessão. Quando um cliente realiza uma requisição de conexão com o servidor, esta pode ou não ser aceita. Se aceita, o servidor irá estabelecer e manter uma conexão com o cliente sob um novo *socket*, livrando o *socket* original para receber requisições de conexão de outros usuários. Algumas características importantes desse tipo de protocolo são: confiabilidade, entrega ordenada e controle de fluxo.

2.3 User Datagram Protocol (UDP)

A ideia do protocolo *UDP* é que o usuário possa enviar seus *datagrams* o mais rápido possível, deixando a encargo da rede entregar os pacotes em um modelo *best-efforts*, despriorizando os *handshakes* e *reliability* em prol de entregar os dados mais recentes. O *UDP* não possui o *overhead* de conexão do *TCP*, o que, em tese, diminuiria o intervalo entre a necessidade de enviar algo e o momento em que de fato este dado é recebido pela aplicação remota.

2.4 Casos de Uso

Muitas aplicações do dia a dia fazem uso da arquitetura *cliente-servidor*, como a troca de e-mails, acesso à internet ou acesso a um banco de dados. O modelo cliente-servidor, tornou-se uma das ideias centrais de computação de rede. A maioria dos servidores possui uma relação *um para muitos(1:n)* com o cliente, ou seja, um único servidor é capaz de prover recursos para muitos clientes ao mesmo tempo.

Aplicações que mais beneficiam-se do protocolo TCP são aquelas que alinham-se com a filosofia do protocolo e também são "orientadas à conexão", sobretudo se além disso precisem de confiabilidade. Por exemplo, quase todos os programas de transmissão de arquivos de dados utilizam o TCP pois devem garantir a integridade da chegada e recebimento dos dados em ordem. Outros exemplos clássicos são os programas *SSH (Secure Shell)* e o *TELNET (Telecomunications Network)*.

Já o *UDP* é usado em aplicações que utilizem-se de suas vantagens de velocidade e não se importam com seus *downsides*, como *streaming* de vídeo ou jogos online. Nesses contextos, caso um pacote antigo seja perdido, ele torna-se obsoleto rapidamente e perde utilidade para o usuário, que está sempre interessado em receber a informação mais recente possível. Essa *unreliability* geralmente é compensada na camada de aplicação com soluções de processamento (como interpolação dos dados de um vídeo ou previsão de movimento) para preencher lacunas de pacotes que foram perdidos.

3 Implementação

Para realizar a implementação da aplicação usou-se [3] como modelo prático, uma vez que o livro apresenta as implementações de *hand-shaking* desejadas de comunicação por TCP e UDP já incluindo tratamento de erros na linguagem C. Já os conceitos teóricos que auxiliaram, sobretudo, no entendimento do funcionamento da aplicação podem ser vistos com mais detalhes em [4]. Um detalhe interessante de implementação pode ser observado na Tabela 1, na qual é possível notar a menor complexidade do servidor *UDP*.

3.1 Clientes

3.2 Cliente TCP

O *cliente*, possui uma implementação mais simples que o *servidor*, visto que, sua única função é fazer uma requisição e aguardar um retorno. Uma vez que trabalhou-se com protocolo *TCP*, em um primeiro momento o *cliente* avisa ao *servidor* que irá abrir uma conexão, o que chamamos de *hand-shaking*. Após estabelecida essa conexão inicial, os *sockets* estarão se comunicando e ambos podem trocar informações.

No cliente, criou-se uma interface na qual o usuário pode escolher qual ação ele deseja executar e então informar os parâmetros necessários para que a execução seja bem sucedida. Para fins deste projeto, utilizamos a função (6), que retorna todas as informações associadas a um email. Então, usando a função *ssize_t send(int socket, const void *buffer, size_t length, int flags)* da biblioteca *sys/socket.h*, o *cliente* envia ao *servidor* o dígito "6"e o e-mail em que deseja executar a consulta. Para isso é preciso informar o socket que está sendo usado, a mensagem a ser enviada e o tamanho da mensagem. Uma nuância, porém, é que não utilizamos diretamente a função *send()*, pois ela não garante o envio de todos os bytes: chamamos a função *send_all()*, que nada mais é que uma repetição de *send()* até todos os bytes serem enviados.

Após requisitar a ação do *servidor*, basta ao *cliente* esperar o processamento dela acontecer, e então receber a resposta. Para receber a mensagem usa-se também uma função da biblioteca *sys/socket.h*, chamada *ssize_t recv(int socket, void *buffer, size_t length, int flags)*, cujos parâmetros são o socket utilizado na conexão, um *buffer* na memória no qual será salva a mensagem recebida e o tamanho dele.

É importante ressaltar que após realizada a comunicação, é o *cliente* o responsável por encerrar a conexão entre ambas as partes.

3.3 Cliente UDP

Tiramos as funcionalidades extras do cliente TCP e deixamos apenas a funcionalidade (6), de requisitar informações de um email. O usuário informará o e-mail via interface e, logo em seguida, já é criado um *socket* e é realizada uma chamada a *int sendto(int sockfd, const void *msg, int len, unsigned int flags, const struct sockaddr *to, socklent tolen* tentando mandar esta requisição ao servidor. Esperamos uma resposta com uma chamada a *int recvfrom(int sockfd, void *buf, int len, unsigned int flags, struct sockaddr *from, int *fromlen*

3.4 Servidor

A implementação do *servidor* é mais complexa e exige um pouco mais de atenção do que a do *cliente*. Ele é o responsável por administrar as informações salvas dos perfis e manusear operações com elas, precisando garantir permanência nos dados.Para isto, optou-se por usar um arquivo texto no qual eram armazenadas as informações. Este, por sua vez, era atualizado ao fim de toda requisição, garantindo que informações alteradas neles ficassem acessíveis aos demais. Uma limitação da abordagem escolhida acontece caso dois clientes diferentes requisitem modificações para um mesmo perfil, visto que a solução não lida com esse tipo de concorrência. Em uma implementação mais robusta, poderia ser utilizado um *Sistema Gerenciador de Banco de Dados (SGBD)* que garantisse operações *ACID (Atômicas, Consistentes, Isoladas e Duráveis).*

3.5 Servidor TCP

O servidor TCP é mais complexo. A explicação de tal complexidade dá-se pois, antes de tratar a requisição do cliente, primeiro é preciso criar um mecanismo que permita atender a vários *clientes* ao mesmo tempo, ou seja, liberando o *socket* de *hand-shaking* no caso do TCP. Para resolver o primeiro problema e criar um servidor concorrente, usou-se a função *pidt fork(void)* da biblioteca *sys/types.h*, ela é responsável por criar um processo filho, semelhante ao processo pai. Desse modo, o processo pai fica responsável por apenas aceitar as conexões que chegam ao *servidor*, e então direcioná-las para um novo processo que irá de fato lidar com ela. Assim como acontecia na implementação do *cliente*, vide secção 3.1, um novo socket era estabelecido para cada nova conexão, e as funções *send()* e *recv()* foram usadas para enviar e receber arquivos entre ambos os lados da aplicação.

3.6 Servidor UDP

O server *UDP* também é mais simples por ter somente uma funcionalidade de consulta. Ao recebermos um contato no *socket*, checamos quem é o cliente e devolvemos a requisição utilizando as mesmas funções de cliente, sem gerar um processo filho.

4 Experimento

4.1 Metodologia

Foram executadas 900 iterações de um script em Python que realizava uma requisição do tipo (6 - Dado o email de um perfil, retornar suas informações) para o TCP e uma normal para o UDP no sistema de perfis:

Medimos o tempo de execução do programa do cliente como o tempo a partir do primeiro byte enviado ao servidor até o último byte recebido. Medimos o tempo de processamento do servidor como sendo o tempo do primeiro byte recebido até o último enviado. Assim, o tempo de comunicação será o tempo do cliente subtraído do tempo do servidor.

Os testes foram executados em com o servidor em um computador na rede do Instituto de Computação UNICAMP e o cliente e um notebook ligado na rede Eduroam.

4.2 Resultados

Os resultados de média, desvio padrão e intervalos de confiança do tempo de execução do cliente usando protocolo *TCP* podem ser vistos na Tabela 2. Os mesmos valores respectivos ao servidor com protocolo *UDP* podem ser vistos na Tabela 3. A fim de obter os resultados apresentados, usou-se conceitos vistos em [2], a partir do qual foi possível obter a fórmula para o cálculo do desvio padrão, que pode ser vista na Equação 1 e também para o cálculo do intervalo de confiança, que para um valor de 95%, é dada pela Equação 2.

Já os histogramas obtidos para as execuções junto ao Servidor *TCP* podem ser vistos na Figura 2a, que mostra o tempo de execução do cliente, e na Figura 2b, que mostra o tempo de comunicação. Enquanto que os histogramas para execução do Servidor *UDP* se encontram na Figura 3a, que mostra o tempo de execução do cliente, e na Figura 3b, que mostra o tempo de comunicação.

5 Discussão

A implementação do sistema nos fez lidar com camadas de "baixo nível"de abstração no *stack* de rede. Tivemos de tratar buffers, sockets e particularidades de confiabilidade do protocolo, algo que normalmente fica escondido para camadas de aplicação. Isto fez nosso código tomar um tamanho considerável mesmo para um projeto simples: o UDP utilizou 238 linhas de código em C e o TCP 329, ambos utilizando várias bibliotecas, inclusive uma nossa com 323 linhas. Portanto, pudemos perceber que a implementação do UDP é bem mais simples que a do TCP.

Além desta facilidade, o UDP mostrou-se 20.94% mais rápido que o TCP, confirmando o que a teoria previa e validando seus casos de uso citados anteriormente.

Não houve perda de pacotes. Conjecturamos que isso tenha ocorrido devido a proximidade física dos dois computadores na rede e ao pequeno volume de bytes do *datagram*.

6 Conclusão

Após nossa análise, concluímos que em um contexto reduzido, de proximidade de rede, baixa latência e pequeno volume de dados, o UDP mostra-se superior em velocidade mantendo *reliability* e com uma implementação mais fácil.

Para validarmos esta conclusão, necessitaríamos realizar testes com uma base de dados mais heterogênea, computadores mais afastados e com conexões instáveis à internet. Sugere-se esta abordagem para uma possível continuação deste trabalho.

Referências

[1] MADEIRA Edmundo. Projeto 2. Disponível em: http://www.ic.unicamp.br/~edmundo/MC833/turma1s2019/proj2_18331s19.pdf, Acesso em: 20-03-2019.

[2] Charles Grinstead and J Laurie Snell. Introduction to probability / charles m. grinstead, j. laurie snell. *SERBIULA (sistema Librum 2.0)*, 04 2019.

[3] Brian Hall. Beej's guide to network programming. using internet sockets. published online by author, 2007.

[4] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach (6th Edition)*. Pearson, 6th edition, 2012.

ANEXOS

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \overline{x})^2}{n-1}} \tag{1}$$

(1) Fórmula para o cálculo do desvio padrão de um espaço amostral.

$$(\overline{x} - 1.96 \cdot \frac{\sigma}{\sqrt{n}}, \overline{x} + 1.96 \cdot \frac{\sigma}{\sqrt{n}}) \tag{2}$$

(2) Fórmula para o cálculo do intervalo de confiança de 95%.

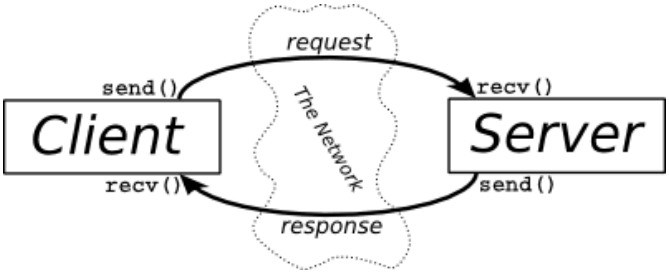
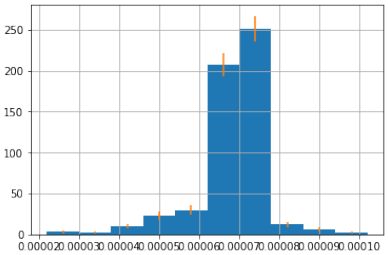
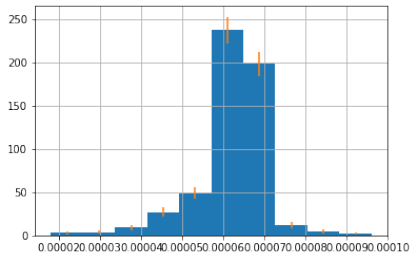


Figura (1) Diagrama cliente servidor.

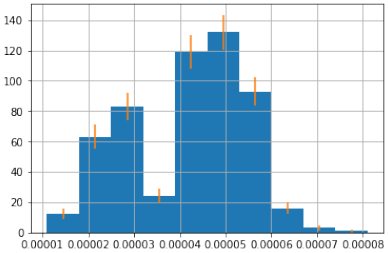


(a) Tempo total do servidor TCP.

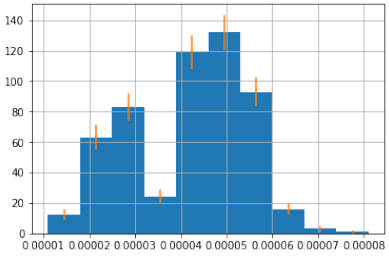


(b) Tempo de comunicação do servidor TCP.

Figura (2) Tempos do servidor TCP.



(a) Tempo total do servidor UDP.



(b) Tempo de comunicação do servidor UDP.

Figura (3) Tempos do servidor UDP.

Tabela (1) Tamanho dos códigos usados

Arquivo	TCP - Linhas	UDP - Linhas
cliente.c	158	112
server.c	164	124
funcoes.c	343	336
funcoes.h	100	100
script de testes	126	92
TOTAL	891	764

Tabela (2) Tempos Para o Servidor TCP

Tempo	Média(μ s)	Desvio(μ s)	Confiança .95(μ s)
Total	68.0	8.22	67.36 - 68.73
Comunicação	62.1	8.29	61.40 - 62.78

Tabela (3) Tempos Para o Servidor UDP

Tempo	Média(μ s)	Desvio(μ s)	Confiança .95(μ s)
Total	49.1	12.89	48.04 - 50.20
Comunicação	42.0	13.01	40.89 - 43.07