MC833

Relatório - Projeto 1

TCP

Marcio Ivan de Oliveira Coelho Filho 173645

Thomas Jun Yamasaki 177677

18 de Abril de 2018

1. Introdução

Para este projeto, implementamos um servidor e um cliente *TCP* em *C*, integrado com um sistema de banco de dados *MYSQL*, com o intuito inicial de aplicar os conceitos de socket e comunicação e uma futura comparação com servidor *UDP*, simulando um sistema de consultas de dados de disciplinas de uma universidade. Analisamos as implementações feitas e o tempo de transmissão entre o servidor e o cliente.

2. Descrição geral e Casos de uso

O servidor *TCP* e o cliente *TCP* foram implementados em *C* com base no código disponibilizado pelo livro *Beej's Guide to Network Programming* e nos códigos vistos em sala de aula.

Basicamente, o servidor executa todos os passos básicos (criação de *socket*, *binding*, *listening*) até chegar no primeiro *loop* principal, em que são executadas as funções *accept()*. Neste ponto é criado um novo processo, usando se a função *fork()*, para que o servidor possa continuar esperando por novas conexões. Tal processo é responsável por mandar informações lidas de um banco, de acordo com o que for pedido pelo cliente. O servidor atende as consultas de um dado cliente, que faz as requisições por meio de determinadas opções possíveis denominadas de *opcodes*, até que ele decida sair do sistema para fazer login em outro tipo de usuário ou mesmo encerrar o programa. Desde o momento do fork, o servidor já fica em *standby* até que receba outro pedido de conexão. A conexão com o banco de dados é feita por usuário, ou seja, podem ser estabelecidas múltiplas conexões para o uso do banco.

Para que o cliente estabeleça a conexão com o servidor (pela porta setada em 8000), ele precisa especificar o hostname (*IP do servidor).* Assim, ele ganha acesso às funcionalidades do banco de consultas.

Inicialmente, criamos o servidor para funcionar exclusivamente em LAN, devendo tanto o cliente quanto o servidor estarem rodando na mesma rede. Entretanto, expandimos a ideia e disponibilizamos o servidor para ser testado via WAN. Para testa-lo, basta que o cliente execute o programa especificando o hostname ‘rabbitnode.stream’.

- **Exemplo de uso:**

```
$ make client                    // Instalação
$ ./client rabbitnode.stream     // Execução
```

Ao iniciar o servidor, o cliente escolhe entre dois possíveis “modos”:

1. Professor
2. Aluno

Ao escolher em qual modo deseja logar, é apresentado ao cliente uma lista de operações (5 em comum entre os dois modos, e uma opção extra para o professor):

0. Logout
1. Listar códigos das disciplinas
2. Buscar ementa
3. Buscar comentário sobre a próxima aula
4. Listar informações de uma disciplina
5. Listar informações de todas as disciplinas
6. Escrever comentário sobre a próxima aula de uma disciplina (Exclusivo do professor)

Basta então o cliente escolher a opção desejada, e será guiado pela interface. Podendo ser solicitado a inserir a disciplina desejada ou o comentário que gostaria de inserir, dependendo de qual operação tiver escolhido.

3. Armazenamento e Estrutura de Dados do Servidor

Para o armazenamento de dados do nosso servidor, utilizamos o *MySQL*, que é integrado com o servidor fazendo uso de bibliotecas prontas (C API). O *MySQL* guarda informações dos tipos de usuários e dos dados que podem ser consultados pelos usuários. As informações que tais usuários podem acessar estão contidas em uma única tabela DISCIPLINAS, que possui campos especificados conforme os requisitos do enunciado. Além de leitura de dados, o banco permite inserção de dados caso o usuário tenha autoridade para tal. A autenticação é feita pelo servidor, e dependendo do usuário que logar no programa, ele pode fazer determinadas consultas.

Para o envio e recebimento de pacotes, escolhemos tratar todos os dados de como “strings” (vetor de char), fazendo as devidas transformações antes (ou depois) quando o dado que deveria ser manipulado seria um numero (caso das opções de login e da seleção de operações). Optamos por esse caminho para possuirmos consistência no comportamento e controle total sobre os dados que estávamos manipulando, pois vimos que certas vezes os envios e recebimentos não ocorriam da maneira que esperávamos quando enviávamos valores diretamente de “int” ou quando alternávamos os envios entre char e “int” repetidas vezes.

Para evitar este problema, criamos uma função de *get_input(),* que utiliza *fgets()* e tratava a “string” recebida, de forma a garantir que não houvesse lixo no buffer quando esta mensagem fosse enviada.

4. Detalhes da implementação do servidor TCP

O servidor TCP comunica-se com o cliente usando as funções *read_buffer()* e *write_buffer()*. Elas foram implementadas de forma a sanar possíveis problemas que poderíamos encontrar na comunicação entre o servidor e cliente, sendo o principal problema o fato de que muitas vezes havia “lixo” no buffer que acaba sendo enviado junto, ou que não era devidamente enviado (ou removido), o que causava comportamentos inesperados do servidor. Além disso, também haviam situações em que os pacotes que deviam ser enviados (e recebidos) poderiam exceder o tamanho limite estipulado por nós (100 bytes).

Para resolver ambos os problemas, utilizamos uma sugestão encontrada no Capitulo 6 do “*Beej's Guide to Network Programming”*, que tratava sobre envios e recebimentos parciais de dados. Antes de enviar de fato um pacote, enviamos um “header” (que escolhemos possuir 4 bytes para prevenir comportamentos inesperados) correspondendo ao tamanho que a mensagem a ser enviada possui.

Assim, nossa função *write_buffer()* chama diversas vezes a função *send()*, até que esteja garantido que toda a mensagem foi enviada. E, do outro lado, a função *read_buffer()* receberá este header, e então chamará *recv()* até ter certeza que o numero de bytes recebido condiz com o tamanho do header.

5. Análise de Tempo

Para a coleta de dados, utilizamos a função de *gettimeofday()* de duas formas: primeiro, isolando uma única troca de comunicação entre o servidor e cliente (para estipularmos um valor médio para o tempo de comunicação) e depois o tempo total de duração para cada operação.

Para a avaliação do tempo de comunicação, focamos em tentar enviar um pacote “cheio”, isto é, contendo o valor máximo aceito pela nossa implementação (100 bytes), fato que pode ter causado uma certa diferença entre o tempo de comunicação que de fato ocorreu entre o servidor e o cliente, e o tempo medido na tabela.

Visando obter uma amostra melhor de dados do tempo, adaptamos as funções das operações para também escreverem em um arquivo de log seu tempo de duração. Com isso, rodamos um script para executar por um breve período uma série de comandos de casos de teste entre o cliente/servidor, para obtermos medidas o suficiente.

Os resultados podem ser conferidos na tabela a seguir:

Rede	Tipo	Medidas	Média	Desvio Padrão	Intervalo de confiança (95%)
WAN	Tempo de comunicacao	2250	0.082966	0.058480	(0.082966 ± 0.002416)
	Operação 1	2137	0.125000	0.126783	(0.125000 ± 0.005375)
	Operação 2	2131	0.158237	0.084901	(0.158237 ± 0.003605)
	Operação 3	2130	0.156476	0.080884	(0.156476 ± 0.003435)
	Operação 4	2128	0.155673	0.078620	(0.155673 ± 0.003340)
	Operação 5	2127	0.157803	0.082208	(0.157803 ± 0.003494)
	Operação 6	2063	0.000518	0.001708	(0.000518 ± 0.000074)
LAN	Tempo de comunicacao	2809	0.008270	0.012575	(0.008270 ± 0.000465)
	Operação 1	893	0.085628	0.043744	(0.085628 ± 0.002869)
	Operação 2	721	0.097516	0.026070	(0.097516 ± 0.001903)
	Operação 3	699	0.104853	0.157766	(0.104853 ± 0.011696)
	Operação 4	932	0.100530	0.040681	(0.100530 ± 0.002612)
	Operação 5	1038	0.083431	0.081882	(0.083431 ± 0.004981)
	Operação 6	3018	0.000736	0.000797	(0.000736 ± 0.000028)

Por meio destes resultados, nota-se que o tempo das operações em uma rede WAN aumenta significativamente (o tempo de comunicação aumentou em quase 10 vezes), fato que era esperado devido ao aumento de trajeto de comunicação, e possíveis complicações durante o processo de roteamento.

Apesar de em boa parte estes resultados condizerem com o que era esperado, houveram algumas disparidades. Em destaque o fato de que a operação 5 (listar todas as informações de todas as matérias) não foi a que gastou maior tempo. Isso provavelmente se deu devido à algum mecanismo do MySQL na hora de realizar as buscas (ou, neste caso, retornar todos os resultados encontrados), além do fato de que estamos tratando com um banco com pouquíssimos dados.

Por fim, outro ponto que vale ser ressaltado é o valor baixo presente na operação 6 (escrita de comentário), o que condiz com o esperado, pois é uma operação que não necessita de um retorno do servidor. Ela apenas envia pacotes fazendo a requisição de atualização).

6. Conclusão

Conseguimos implementar o servidor e o cliente com sucesso, mesmo encontrando vários problemas na leitura e envio de dados, que foram esclarecidos e resolvidos com êxito. Por meio deles, conseguimos observar as dificuldades de se lidar com um buffer e enviar inputs do cliente para o servidor em *runtime*. Aplicamos os conceitos de socket vistos em aula, o que proporcionou uma consolidação de nosso conhecimento e um aprofundamento sobre o tema, especialmente no que diz respeito à comunicação persistente de cliente/servidor, com o servidor sendo capaz de se comunicar com diversos clientes “ao mesmo tempo”.

7. Referências

1. *http://beej.us/guide/bgnet/ (Beej's Guide to Network Programming)*
2. *http://www.ic.unicamp.br/~edmundo/MC833/mc833/SocketTCP1S2018_01.pdf*
3. *http://www.csd.uoc.gr/~hy556/material/tutorials/cs556-3rd-tutorial.pdf*
4. *http://www.cs.rpi.edu/~moorthy/Courses/os98/Pgms/socket.html*