Universidade Federal de Minas Gerais

Departamento de Ciência da Computação

Curso de Redes 2020/2º

Relatório: Trabalho Prático - Aplicação Publish/Subscribe

Aluno: Marcos Vinicius Moreira Santos

Introdução

O código foi dividido em 2 modulos, uma biblioteca de código que está em localizada em src/netowrk que abstrai um servidor e cliente TCP e que foi usada para a implementação da aplicação Publish/Subscribe permitindo que as regras de negócio da aplicação fiquem mais limpas e de fácil compreensão, uma biblioteca localizada em src/protocol que contém um lexer e parser para o protocolo, e o código da aplicação em src/.

Network:

Introdução:

Os arquivos localizados em src/network são os arquivos que abstraem a implementação de uma estrutura de servidor e cliente para comunicação sobre a network utilizando o protocolo TCP/IP. Além disso, também foi implementado estruturas de dados para programação assíncrona como threads e mutexes. A biblioteca foi construida utilizando UNIX sockets e POSIX threads além da biblioteca padrão do C99.

Servidor TCP:

Em src/network/tcp_server.c foi definida uma estrutura de dados e metodos que abstraem um servidor TCP. A abstração em questão utiliza a thread principal da aplicação para esperar por conexões, e quando uma solicitação de conexão é feita por um cliente, essa mesma thread é responsável por criar uma nova thread que irá ser responsável por manter a conexão com o cliente ativa, qualquer mensagem enviada pelo cliente portanto será recebida e tratada dentro dessa thread.

Em caso de aplicações em que a ordem que o servidor irá processar as requisições de diferentes clientes importa, é possivel instanciar o servidor tcp com a flag **TCP_SERVER_SYNC**, essa flag fará com o que servidor execute a primeira requisição que chegar antes que a thread de um outro cliente com requisição pendente comece sua execução.

O comportamento do servidor é definido utilizando-se o método:

```
void tcp_server_t_set_request_handler(struct tcp_server_t* server,
tcp_server_t_request_handler handler);
```

Este método irá definir como o servidor deverá processar uma mensagem qualquer recebida de algum cliente. Uma mensagem so é repassada ao handler quando o servidor detectar um '\n', se essa condição não ocorrerm o server continuará esperando uma nova mensagem que será tratada como a continuação da mensagem anterior.

As estruturas request_t e reply_t são estruturas que armazenam informações uteis como o cliente que fez a requisição, o servidor que está processando a requisição, a mensagem de payload e o tamanho da mensagem enviada pelo cliente.

Cliente TCP:

Foi implementado em src/network/tcp_client.c uma estrutura que abstrai a funcionalidade de um client TCP.

No cliente não foi necessário nenhuma thread extra, foi escolhido a implementação de um cliente que utiliza *'non blocking sockets'*. Essa escolha foi tomada para permitir que o programador utilizando o cliente possa fazer o uso dos recursos computacionais da forma que ele achar melhor baseado na aplicação sendo desenvolvida.

Detalhes de implementação:

Para a implementação das estruturas do servidor, foram utilizadas estruturas de dados assincronas, como threads e mutexes, ambas essas estruturas foram abstraidas utilizando a biblioteca pthreads em src/network/async.c.

Foi feito o uso de UNIX sockets para a implementação do servidor e cliente assim como outras classes da biblioteca padrão como strings, stdlib e stdio. No caso do cliente foi utilizado um *'non blocking socket'* atualizando suas flags para incluir a flag ***O_NONBLOCK*** após sua inicialização.

O uso de threads foi necessário para garantir que um servidor possa servir a mais de um cliente por vez, portanto para cada cliente o servidor cria uma thread unica que será responsável por escutar as mensagens enviadas por esse cliente.

Para armazenar as conexoes ativas com o servidor, foi utilizado uma estrutura de lista encadeada que pode ser utilizada simultaneamente por multiplas threads, sincronizada através do uso de mutexes, onde cada nó armazena as informações references à uma conexão/cliente.

Aplicação Publish/Subscribe:

Toda a lógica relacionada à aplicação de Publish/Subscribe está definida sobre os arquivos *.c e *.h dentro do diretório src/.

A aplicação solicitada define quatro ações de usuário:

1. Cadastrar o cliente em uma tag enviando uma mengagem iniciada com o caractere '+'.

2. Descadastar o cliente de uma tag enviando uma mengagem iniciada com o caractere '-'.

3. Encerrar a execução do servidor e todas as suas conexoes enviando uma mensagem "##kill".

4. Enviar uma mensagem de texto para o servidor que irá redirecionar a mesma mensagem para qualquer cliente que esteja interessado utilizando a sintaxe *"mensagem #tag"*. Essa ação pode ser realizada enviando uma mensagem que não se encaixa em nenhuma das condições das ações anteriores.

Protocolo:

A descrição do trabalho define:

```
Servidores e clientes trocam mensagens curtas de até 500 bytes usando o
protocolo TCP.  Mensagens carregam texto codificado segundo a tabela
ASCII. Apenas letras, números, os caracteres de pontuação ,.?!:;+-*/=@#$%
()[]{} e espaços podem ser transmitidos. (Caracteres acentuados não podem
ser transmitidos.)
```

Ou seja, mensagens devem ser codificadas de acordo com a tabela ASCII. A partir disso, foi entendido que o intervalo de characteres deveria estar entre (48, 57) que abrage os numeros de 0 a 9, (65, 90) que abrange os caracteres maiusculos, (97,122) que abrange os caracteres minusculos e o conjunto de caracteres ,.?!:;+-*/=@#$%()[]{}. Se uma mensagem for enviada e não obedecer esses parametros o servidor irá desconectar o cliente que abortará sua execução.

Outra requisição do trabalho é:

1) As mensagens de interesse (+tag) e desinteresse (-tag) devem ter o sinal (+ ou -) no primeiro caractere e apenas um tag, sem nenhum texto adicional.

2) Tags e especificações de interesse e desinteresse devem estar precedidas e sucedidas por espaço, início da mensagem, ou término da mensagem. Em outras palavras, um string como “#dota#overwatch” não será considerado como tags.

A primeira sentença define que mensagens de interesse e desinteresse devem conter (+|-) no primeiro caractere. Porém, a segunda sentença diz que as especificações de interesse e desinteresse devem estar precedidas e sucedidas por expaço, ou seja, o primeiro caractere poderia ser um espaço seguido de um (+|-), portanto temos duas declarações conflitantes. Foi escolhido a primeira sobre a segunda por ser mais especifica.

Foi assumido que as sentenças " #dota#overwatch" e " #dota#overwatch " estão de acordo com o protocolo e enviam as mensagem " #dota#overwatch" e " #dota#overwatch " na tag "dota#overwatch", porém como solicitado, a sentença "#dota#overwatch" não é contem uma tag válida pois não há espaços antes do primeiro "#" e portanto essa mensagem deverá ser interpretada como uma palavra.

A partir disso, para o desenvolvimento do parser para o protocolo, foi utilizado a seguint Gramática Libre de Contexto:

```
S -> (M*( \#'P)\* M\*)\* | +P | -P | P;
Q -> , | . | ? | ! | : | ; | + | - | * | / | = | @ | # | $ | $ | % | ( | )
| { | };
L -> [a-z] | [A-z] | [0-9];
P -> L* | PQP;
M -> ESPAÇO\*P(ESPAÇO\*P)*;
```

Onde ESPAÇO equivale à um espaço (' '), S ao Protocolo e representa a variavel inicial da gramática, Q aos possiveis sinais, L às possíveis letras, P às palavras e M às frases.

Foi definido em src/protocol um lexer e parser que seguem essa gramárica.

O Servidor:

O servidor é responsável por tratar as requisições feitas pelos clientes. Foi definido um handler, adicionado ao servidor através do método:

```
void tcp_server_t_set_request_handler(struct tcp_server_t* server,
tcp_server_t_request_handler handler);
```

Por sua vez, esse handler verifica a mensagem enviada pelo cliente e, baseado nas quatro condições descritas acima, decide alguma ação que será executado sobre a mensagem.

O servidor utiliza apenas duas estruturas de dados auxiliares, um mapa do tipo 'string'->'lista de inteiros' que irá armazenar em qual tag('string') um determinado grupo de clientes('lista de inteiros') estão interessados, chamaremos essa estrutura de tabela de tags, e uma estrutura de tabela hash que armazenará inteiros e que será utilizada para armazenar quais os clientes que já receberam uma dada mensagem para evitar que clientes cadastrados em duas tags diferentes recebam uma mensagem duplicada em caso de essa mesma mensagem ter sido enviada para ambos as tags.

O Cliente:

O cliente é responsável por 2 coisas:

1. Receber a entrada do usuário
2. Enviar a entrada do usuário para o servidor.
3. Receber as mensagens do servidor

Para receber a entrada do usuário, foi implementada em src/terminal.h algumas funções uteis no manejo de entradas de usuário pelo terminal, como funções não bloqueantes que verificam se existem alguma entrada pendente. A partir disso, a lógica do cliente passa a ser bastante simples, basta testar se existe entrada pendente do usuário, caso exista, a mensagem é enviada ao servidor. Em caso de nenhuma entrada pendente, o cliente tenta ler mensagens enviadas pelo servidor.

Conclusão:

Durante a implementação desse Trabalho foi implementado um Servidor e Cliente que utilizam o protocolo TCP/IP, o que possibilitou entender com mais detalhes o funcionamento deste Protocolo.

Também foi implementado uma aplicação por cima do servidor e cliente TCP abstraidos dentro de src/network que se resume a uma aplicação Publish/Subscribe. Durante a implementação desta aplicação foi possível o melhor entendimento das aplicações que utilizam comunicação sobre a netowrk, e em especial, foi possivel obter "insights" sobre o funcionamento de sistemas como Apache Kafka e RabbitMQ.