Trabalho Final - Relatório da Parte 1

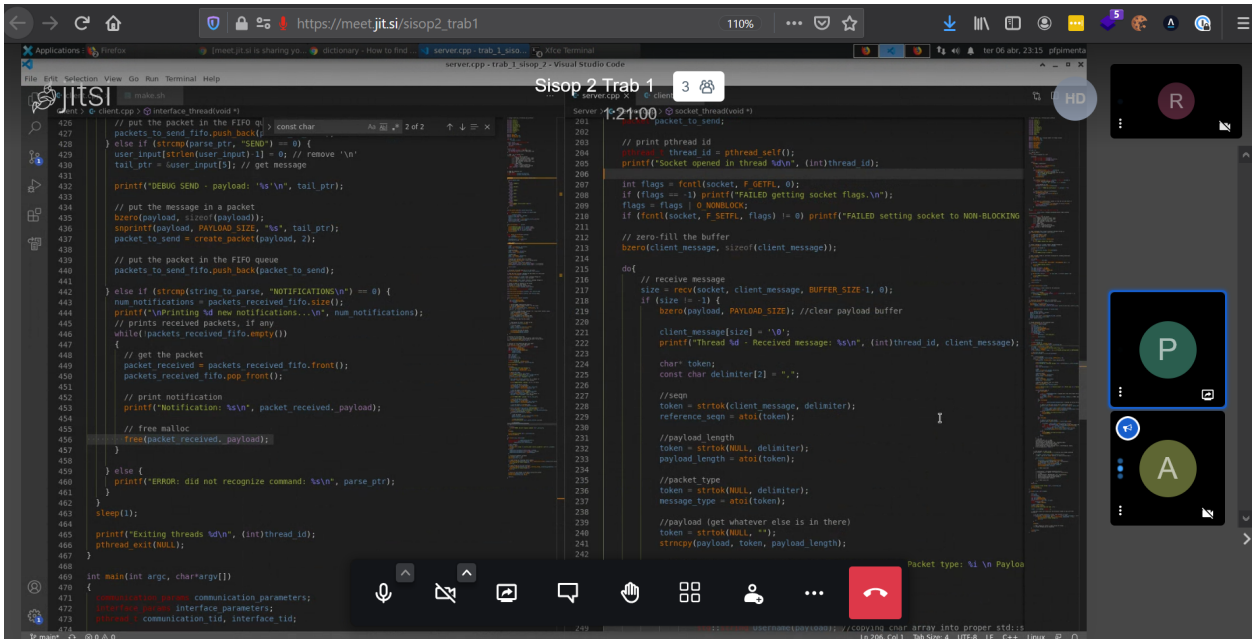
INF01151 - Sistemas Operacionais II N

André Carini afcarini@inf.ufrgs.br
Pedro Foletto Pimenta pfpimenta@inf.ufrgs.br
Roberta Robert rrobert@inf.ufrgs.br

Introdução

Neste trabalho implementamos um sistema de mensagens nos moldes do Twitter, utilizando os conceitos de programação com primitivas básicas de controle de fluxo de mensagens e persistência entre cliente-servidor pelo protocolo TCP, assim como conceitos sobre atomicidade de funções, trechos de código crítico e programação de mutexes de compartilhamento e mutexes de acesso exclusivo.

Como metodologia do projeto, em um primeiro momento trabalhamos de forma assíncrona, organizado em pequenas tarefas. Porém, como boa parte do tempo era utilizada em trechos de código que eram interdependentes, acabamos optando pelo método de trabalho *pair programming* que se mostrou mais efetivo.



Descobrimos principalmente, durante o trabalho, que *pair programming* não só é muito melhor do ponto de vista de qualidade final do código e documentação como também economizou tempo que seria gasto em rebases problemáticos. Foi muito mais rápido para conseguirmos programar os trechos mais críticos do trabalho (viramos super fãs).

Especificações do ambiente de testes

- Versão do sistema operacional e distribuição: Ubuntu Studio 20.04.2 LTS
- Configuração da máquina:
    - 8 GB de memória RAM
    - CPU: Intel(R) Core(TM) i5-6600 CPU @ 3.30GHz
- Compiladores utilizados: g++ (Ubuntu 9.2.1-9ubuntu2) 9.2.1 20191008

Concorrência do servidor com múltiplos clientes

Quando o cliente se conecta ao servidor, é verificado se o limite de sessões simultâneas foi excedido: se estiver dentro do limite, o servidor inicia uma **sessão** e incrementa o contador de sessões daquele usuário.

Para cada sessão que o cliente iniciar, o servidor gera uma nova thread que faz o tratamento de toda a comunicação entre o servidor e aquele cliente. Essa thread fica enviando notificações pendentes e também fica interpretando os pacotes recebidos. As informações são adequadamente registradas e acessadas na *master_table*, nossa estrutura globalmente acessível — mas protegida com *mutexes* apropriados — que guarda o registro de usuários, seus seguidores e suas notificações pendentes.

Conforme as notificações vão sendo enviadas, a lista de notificações pendentes é esvaziada. Caso haja mais de uma sessão ativa para um dado cliente, essa notificação é enviada em todas as sessões ativas desse cliente. Para isso, foi criada uma variável de controle *notification_delivered*, que indica se alguma das threads já enviou a mensagem. Já que temos no máximo sempre duas threads (por usuário), se uma delas já enviou a mensagem a outra pode apagar esta mensagem da lista de mensagens pendentes sem problemas, visto que a mensagem já vai estar entregue para todas as sessões abertas do usuário respectivo.

Áreas de sincronização para acesso dos dados

Foi necessário o uso de mecanismos de sincronização tanto no código do cliente quanto no código do servidor.

No **servidor**, os nossos trechos de código crítico estão em todos os lugares onde lemos ou alteramos a nossa *master_table*, que é uma estrutura de dados global compartilhada por todas as threads. Essa estrutura será descrita na próxima sessão do relatório (*Nossas estruturas*).

Nas partes onde há alteração de alguma informação da *master_table*, utilizamos um mutex de acesso exclusivo. Assim, a tabela só pode ser alterada por uma thread se nenhuma outra thread estiver alterando ou lendo ela. Isso acontece nas seguintes situações:

- Quando uma sessão é iniciada. Nesse momento, a contagem de sessões ativas do usuário em questão é incrementada.
- Quando uma sessão é terminada. Nesse momento, a contagem de sessões ativas do usuário em questão é decrementada.
- Quando um usuário segue um outro usuário. Nesse momento, a tabela é atualizada pois um novo seguidor é adicionado a um certo usuário.
- Quando uma nova notificação a ser reenviada pelo servidor é recebida de um cliente. Nesse momento, a tabela é atualizada pois uma nova notificação é adicionada a cada seguidor do usuário que enviou a notificação.
- Quando a *master_table* é carregada, na inicialização do programa. No momento, esse mutex não é necessário, já que não existe perigo de conflito nesse momento da execução do programa. Adicionamos o mutex nesse momento para evitar problemas futuros, caso a função de carregar a tabela seja usada em outras partes do programa na segunda parte do trabalho.

Nas partes onde há leitura (sem alteração) de informações da *master_table*, utilizamos um mutex de acesso compartilhado. Assim, desde que não haja nenhuma thread alterando a tabela, várias threads podem ler as informações dela concorrentemente. Isso acontece nas seguintes situações:

- Na conexão de um cliente novo, quando é verificado se já existem duas sessões deste usuário. Nesse momento é feita a leitura da variável *num_active_sessions*, pertencente à master_table.
- Quando os seguidores de um certo usuário são recuperados. Isso acontece em duas situações: quando a *master_table* é atualizada e salva na memória e quando uma mensagem nova chega ao servidor, o que causa uma atualização nas listas de mensagens a serem enviadas de cada usuário que segue o usuário que mandou a mensagem.
- Quando é verificado se há novas notificações a serem enviadas para um certo usuário. Isso é feito acessando a variável *messages_to_receive*.
- Quando é recuperada da lista *messages_to_receive* uma notificação a ser enviada para um certo usuário.
- Quando as informações de um usuário são recuperadas da tabela.

Para a implementação desses *mutexes*, usamos como inspiração o pseudo-código dos slides da aula de semáforos, que também lidava com uma situação de escrita e leitura de dados compartilhados entre threads (também conhecida como situação de *Leitores-Escritores*).

No **cliente**, foi preciso garantir o acesso exclusivo nos momentos em que as threads guardam ou recuperam dados das filas FIFO *packets_to_send_fifo* e *packets_received_fifo*. Essas filas serão descritas na próxima sessão do relatório (*Nossas estruturas*). Esses acessos acontecem nas seguintes situações:

- Quando o cliente recebe notificações do servidor e as guarda na fila *packets_received_fifo.*
- Quando o usuário digita uma nova mensagem (SEND) ou solicitação de FOLLOW, que é então guardada na fila *packets_to_send_fifo.*
- Quando o usuário escolhe ver as notificações novas que chegaram, que estão guardadas na fila *packets_received_fifo.* Nesse momento todos os packets guardados nessa estrutura são consumidos (ou seja, lidos e deletados da FIFO) pela thread que lida com a interface com o usuário.
- Quando o cliente envia uma notificação que está na fila *packets_to_send_fifo*. Para isso, é necessário consumir (ou seja, ler e deletar da FIFO) um *packet*.

No caso do cliente, ao contrário do servidor, um simples mutex já basta, pois todas as operações sobre os dados compartilhados entre threads modificam os dados. Logo, sempre é necessário acesso exclusivo.

Tanto no código do cliente como no código do servidor, também utilizamos mutex'es para proteger o acesso e modificação da variável *termination_signal*, usada na lógica de hooks on exit, descrito na seção abaixo.

Nossas principais estruturas e funções

Master_table*:*
Essa estrutura foi implementada com um *map* e contém uma *Row* para cada usuário que já se conectou ao servidor. Essa estrutura *Row* foi implementada como uma classe e contém as seguintes informações sobre cada usuário:

- Quantidade de sessões ativas desse usuário.
- Lista dos usuários que seguem esse usuário.
- Lista de mensagens a serem enviadas a esse usuário.
- Variável de controle de envio de notificação.

Além disso, a classe *Row* também contém métodos para ler e modificar esses campos. Os mecanismos de sincronização foram implementados dentro desses métodos, garantindo assim a segurança deles.

Mutexes*:*
De acordo com o que foi descrito na seção anterior, no servidor foi implementado um mutex de acesso compartilhado para leitura da estrutura compartilhada *master_table* junto com um mutex de acesso exclusivo para situações de modificação dessa estrutura. No cliente, foi implementado dois mutexes de acesso exclusivo para situações de produzir e consumir *packets* das FIFOs *packets_to_send_fifo* e *packets_received_fifo*, descritas logo abaixo.

packets_to_send_fifo e packets_received_fifo*:*
Estas são as estruturas de fila de funcionamento FIFO para que as duas threads do cliente possam passar dados de uma para outra.

A **packets_received_fifo** contém os *packets* que são recebidos pela thread de comunicação, que envia e recebe packets do servidor. A thread de interface, que interage com o usuário através do terminal, consome esses packets da fila FIFO sempre que o usuário executa o comando de ler as novas notificações.
A **packets_to_send_fifo** contém os *packets* a serem enviados para o servidor. Esses packets são criados a cada vez que o usuário digita o comando que cria uma nova mensagem a ser enviada. Nesse momento, a thread de interface põe o packet criado na fila FIFO. A thread de comunicação, então, detectará que existe uma nova mensagem a enviar, consumirá o packet da fila FIFO, e o enviará.

Structs de packet e communication_parameters*:*
Structs da base de comunicação e mensageria entre cliente e servidor do trabalho, que foram seguidas de acordo com a sugestão da especificação.

```c
typedef struct __packet{
    uint16_t type; // Tipo do pacote:
        // 0 - CONNECT (username_to_login, seqn)
        // 1 - FOLLOW (username_to_follow, seqn)
        // 2 - SEND (message_to_send, seqn)
        // 3 - MSG (username, message_sent, seqn)
        // 4 - ACK (seqn)
        // 5 - ERROR (seqn)
    uint16_t seqn; // Número de sequência
    uint16_t length; // Comprimento do payload
    char* _payload; // Dados da mensagem
} packet;
```

```c
typedef struct __communication_params{
  char* profile_name;
  char* server_ip_address;
  int port;
} communication_params;
```

Função de controle do servidor - void * socket_thread(void *arg)
Função principal de manipulação e controle das threads de conexão/sessão com os clientes. Cada cliente que se conecta ao servidor faz com que uma thread no servidor seja criada, executando essa função.
Ela faz toda a lógica de tratamento de sessões abertas pelo cliente, além de toda a lógica do laço que classifica, pelo *message_type*, os pacotes recebidos como pacote de tipo *SEND*, *FOLLOW*, *CONNECT*, or *DISCONNECT*. Ela também implementa a lógica de envio de notificações para os clientes.
Ela é baseada em um loop que, alternadamente, verifica se existe alguma notificação a ser enviada para o cliente associado, enviando-as se for o caso, e controla o recebimento de mensagens desse cliente.
Essa função também tem a importante responsabilidade de gerenciar a *master_table*, atualizando-a quando necessário e salvando um backup dela a cada atualização. Ela também garante que mensagens a ser enviadas para usuários que estejam offline sejam entregues no momento que o tal usuário se conectar ao servidor.

Funções de comunicação do cliente - void * communication_thread(void *arg) e void communication_loop(int socketfd)
Essas funções controlam o recebimento de notificações do servidor e o envio de mensagens ao servidor. Essa parte é baseada em um loop que, alternadamente, verifica se existe alguma mensagem a ser enviada na fila FIFO **packets_to_send_fifo**, enviando-a se for o caso, e controla o recebimento de notificações do servidor. Para poder alternar entre o recebimento e

envio de mensagens ao servidor, setamos o socket como não-bloqueante. Assim, se não houver nenhuma mensagem recebida no socket, o programa segue a sua execução.

Funções de interface do cliente - void * interface_thread(void *arg)
Essa função gerencia a interação do usuário, dando suporte aos comandos possíveis do usuário. Essa parte é baseada em um loop que lê e reconhece um input de comando do usuário. Existem 4 situações possíveis:

1. Comando FOLLOW, que cria e envia ao servidor um pacote do tipo FOLLOW.
2. Comando SEND, que cria e envia ao servidor um pacote do tipo SEND.
3. Comando NOTIFICATIONS, que faz com que as novas notificações recebidas do servidor sejam printadas na tela.
4. Se o comando digitado é inválido, o usuário é informado.

Hooks on exit, Cntrl + C e Cntrl + D
Foi feito, através da captura do buffer de teclado (ctrl+D na aplicação do cliente) e de handler de sinais (no servidor e no cliente: SIGINT, SIGABRT, SIGTERM) os controles de fim de fluxo. Foi criada uma flag *termination_signal* para ter-se controle dos fluxos e também acabar os processos das threads através da diretiva de *pthread_cancel*, de acordo com cada tipo de terminação.

Primitivas de comunicação utilizadas

int socket(int *domain*, **int** *type*, **int** *protocol*);
int bind(int *sockfd*, **struct sockaddr** **addr*, **socklen_t** *addrlen*);
Utilizados pelo cliente e pelo servidor para preparar um socket para conexões.

int listen(int *sockfd*, **int** *backlog*);
Utilizado pelo servidor para preparar o socket no formato *LISTEN* para poder aceitar conexões do cliente.

int connect(int *sockfd*, **struct sockaddr** **addr*, **socklen_t** *addrlen*);
Utilizado pelo cliente para iniciar uma conexão com o servidor.

int accept(int *sockfd*, **struct sockaddr** *addr*, **socklen_t** *addrlen*);
Utilizado pelo servidor para aceitar uma conexão pendente de um cliente em um socket *LISTEN*, gerando um novo socket ativo que encaminhamos para uma nova thread que tratará das comunicações com o cliente em questão.

ssize_t recv(int *sockfd*, **void** **buf*, **size_t** *len*, **int** *flags*);
ssize_t write(int *fd*, **const void** **buf*, **size_t** *count*);
Utilizados pelo cliente e pelo servidor para enviar e receber dados através do socket.

Dificuldades encontradas durante o desenvolvimento

Tivemos alguns problemas durante o desenvolvimento do trabalho, que vamos descrever brevemente abaixo:

- Alguns erros de cálculo ao alocar espaço para buffers, *causando segmentation faults* que foram trabalhosos para identificar (aprendemos a usar o gdb!)

- Problemas com casting e tipagem: usamos muitas vezes os tipos nativos de C, mas as vezes também tipos de C++. Como não tínhamos muita experiência em C++, perdemos mais tempo do que o esperado com esses problemas.

- Descompasso na tradução da especificação do trabalho: tivemos várias discussões internas sobre a interpretação da especificação do trabalho (principalmente da parte de notificações pendentes) e também do modo que implementaríamos cada funcionalidade.