Universidade Federal do Rio Grande do Sul **Instituto de Informática**
Professor Alberto Egon Schaeffer Filho **alberto@inf.ufrgs.br**
INF01120 Sistemas Operacionais II N **Turma B (2021/1)**

Trabalho Prático (Parte 1)

Iron P. da Silva (00231590)
Luiza Schmidt (00281954)
Mayra Cademartori (00301639)
Tiago S. Ceccon (00278142)

29 de setembro de 2021

1 Especificações

Durante o desenvolvimento e testes foi utilizado o sistema operacional Linux/Ubuntu com compilador GCC 7.5.0. Para gerar os makefiles e compilar foi utilizado CMake 3.21.3, e para a realização de alguns testes utilizamos gtest.

2 Implementação

A implementação dos requisitos da aula prática demonstrou o uso prático das funcionalidades básicas de comunicação por sockets UNIX em C++. O ambiente foi configurado em máquina local e suas dependências importadas automaticamente. O código foi dividido em módulos de bibliotecas: tendo a do Cliente, do Servidor, a biblioteca comum e uma third party que foi utilizada apenas para fazer o parsing dos argumentos passados pela linha de comando ao inicializar os processos.

2.1 Funcionalidades

Ao iniciar o server pela primeira vez ele cria automaticamente uma database vazia. O formato de database que utilizamos é um arquivo de texto que vai sendo preenchido no formato "username,follower1,follower2,follower3, ... ;". Cada vez que um usuário novo faz login pela primeira vez uma linha é criada para ele, e cada vez que um usuário segue outro a linha do usuário a ser seguido é atualizada adicionando o novo seguidor.
Um mesmo usuário pode conectar em no máximo duas sessões concomitantes e no caso de mais tentativas enquanto já possua duas sessões ativas ele recebe uma mensagem avisando que não foi possível conectar. No processo servidor, um "heartbeat"é mostrado em tela a cada 5 segundos pra acompanhar a execução.
É possível para um usuário além de login realizar outras ações. Entre elas:

1. `SEND`: Utilizado para enviar mensagens de no mínimo um caractere aos seguidores de um usuário.

2. `FOLLOW`: Segue usuários existentes na database. O username pertencente à pessoa seguida é adicionado a database após o registro do usuário que seguiu, ou seja, se Felipe seguiu Amanda, em uma linha teríamos `"Amanda,Felipe;"`. Desse momento em diante, o usuário que realizou esse comando passa a receber as novas notificações geradas pelo usuário a ser seguido.

3. `CLOSE`: Encerra a sessão do usuário. O usuário receberá as mensagens enviadas pelos usuários que segue no período em que esteve completamente offline (i.e., nenhuma sessão ativa) quando realizar login novamente. Além de enviar o comando "CLOSE", usuário pode também realizar essa ação pressionando Ctrl+C ou Ctrl+D. Quando o servidor é fechado todas as sessões abertas são encerradas graciosamente e os usuários avisados.

Para realizar a ação desejada, o usuário deve digitar o nome dela, seguido pelo argumento relevante. Tratamos as ações de forma case-insensitive, portanto tanto "FOLLOW ufrgs"quanto "follow ufrgs"geram o mesmo resultado, por exemplo (no caso, fazem o usuário atual começar a seguir o usuário "ufrgs").

2.2 Concorrência

O servidor se encontra em loop realizando a operação de `accept` em seu socket que está em estado de listening. Ao obter uma conexão é lançada uma thread que trata a conexão do cliente com o socket file descriptor obtido.
As threads de conexão com clientes são armazenadas em um vetor para posterior chamada de `join`, se aplicável.
O servidor adiciona o cliente em sua database de perfis caso não exista ou o carrega da database se este existe. Então um objeto de sessão vinculado ao perfil é criado ou, caso já exista, é modificado para contabilizar a nova sessão.
No lado cliente, são utilizadas duas threads: uma para manipular dados recebidos do servidor e uma para manipular comandos recebidos do cliente.

2.3 Connection Manager

Os objetos `Connection Manager` do Cliente e Servidor tiveram implementações semelhantes. Ambos recebem uma entrada resultante to *parsing* dos parâmetros do programa deve ser fornecida ao seu construtor e ambos fazem a criação e *binding* de sockets. As diferenças relevantes entre as versões Server e Client de `Connection Manager` incluem: no manager do Server, o método `getConnection()` é responsável por aceitar conexões entrantes e também possui uma variável de backlog com um limite estipulado de conexões para a fila; já no manager do cliente, existe um timeout para limitar a demora de transferência de pacote de dados já na criação do socket TCP. Serão especificadas as implementações em seções seguintes. Abaixo um recorte da definição do `ServerConnectionManager`:

```
class ServerConnectionManager {
 public:
   /*
    * Creates a listening port
    * Exits with error if it fails
   */
   ServerConnectionManager(const cxxopts::ParseResult& input);
```

```
    ~ServerConnectionManager();

    /*
     * Returns a socket file descriptor
     */
    int getConnection();

    static void closeConnection(int sfd);
    ...
};
```

2.4 Sincronização e controle

Apenas semáforos foram utilizados com o propósito de sincronização e execução atômica de seções críticas enaqueles métodos em que são necessários. Sua implementação reflete a implementação comum explicada durante o semestre. Inicializamos o `_count` do semáforo com um inteiro e, ao chamar `wait()`, caso a variável `_count` seja zero, bloqueamos a thread até que uma variável de condição seja satisfeita e essa seja desperta por outra thread através do método `notify()`.

```
class Semaphore {
 public:
    Semaphore (int count = 0);
    ~Semaphore();

    void wait();
    void notify();

 private:
    std::mutex _mutex;
    std::condition_variable _condition;
    int _count = 0;
};
```

Abaixo um exemplo de uso dos métodos `wait()` e `notify()` de um semáforo da classe `PersistentUser`, de forma a garantir atomicidade e evitar concorrência ao acessar o banco de dados através da interface do usuário. Também foram usados semáforos para realizar ações: manejar threads e processar notificações para os seguidores de um usuário que mandou uma mensagem, processar mensagens recebidas de um usuário seguido, seguir usuário, ler e escrever do buffer de mensagens, criar e fechar sessões de usuário, entre outras.

```
void PersistentUser::addFollower(std::string follower) {
    _sem.wait();

    _followers.insert(follower);
```

```
    _pm.saveUser(*this);

    _sem.notify();
}
```

Temos também uma biblioteca Stoppable que é utilizada tanto no servidor quanto no cliente. Seu único objetivo é gerenciar uma variável booleana atômica a nível de processo para que todas as threads saiam de seus loops infinitos.

2.5 Comunicação

As funções estáticas `dataReceive` e `dataSend` presentes nas estruturas de Connection Manager são responsáveis pela comunicação de dados.

```
class ServerConnectionManager {
    ...
    static ssize_t dataSend(int sfd, PacketData::packet_t packet);
    static ssize_t dataReceive(int sfd, PacketData::packet_t& packet);
    ...
};
```

Os tipos de um pacote são: `LOGIN`, `MESSAGE`, `FOLLOW`, `CLOSE`, `SUCCESS`, `ERROR` ou `NOTHING`.

A estrutura dos packets é demonstrada no código abaixo e seu uso é explorado em seções seguintes. Utilizamos o `timestamp` de criação do pacote, `payload` com até 129 caracteres (considerando o tamanho da mensagem mais o caractere final) e `extra` para enviar os nomes dos usuários. O atributo `packed` está sendo usado para garantir que durante a compilação haverá alinhamento de 8 em 8 bytes.

```
    typedef struct __attribute__((packed))s_packet {
        packet_type type;
        uint16_t seqn;
        uint16_t length;
        uint64_t timestamp;
        char payload[PT_PAYLOAD_SIZE];
        char extra[PT_EXTRA_SIZE];
    } packet_t ;
};
```

2.6 Principais funções

2.6.1 ServerConnectionManager

O `ServerConnectionManager` realiza o *bind* do Server em uma porta conhecida (ou sai com mensagem de erro caso tenha falhado).

```
ServerConnectionManager::ServerConnectionManager(const cxxopts::ParseResult&
    input) {
    _port = std::to_string(input["port"].as<unsigned short>());
    _bindListeningSocket();
}
};
```

Configuramos a conexão TCP ao chamar `_bindListeningSocket()`.

```
void ServerConnectionManager::_bindListeningSocket() {
    struct addrinfo hints, *addrs = NULL;
    int err = 0;

    // Get address
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;
    hints.ai_protocol = IPPROTO_TCP;

    err = getaddrinfo (
        NULL,
        _port.c_str(),
        &hints,
        &addrs
    );
    ...
```

Abaixo, o código onde os endereços disponíveis são percorridos e o *binding* do socket ao endereço é realizado. Caso o *binding* tenha sucesso começamos a escutar possíveis conexões.

```
    for (struct addrinfo* addrptr = addrs; addrptr != NULL; addrptr =
        addrptr->ai_next) {

        if (bind(_socketFileDesc, addrptr->ai_addr, addrptr->ai_addrlen) == 0)
            break;
    }
    ...

    err = listen(_socketFileDesc, _backlog);
```

Abaixo código da aquisição de conexão. A função busca aceitar uma nova conexão.

Para evitar bloqueio indeterminado da thread principal do servidor, um sinal de alarme é configurado. Ao aceitar uma nova conexão, uma nova instância de Socket File Descriptor (`newSFD`) é retornada. Esse processo se repete em laço na função `main`.

```
int ServerConnectionManager::getConnection() {
    sockaddr_storage client_addr;
    socklen_t client_addr_size = sizeof(client_addr);

    alarm(5);
    // Socket File Descriptor
    int newSFD = accept(_socketFileDesc, (sockaddr*) &client_addr,
        &client_addr_size);

    alarm(0);

    return newSFD;
}
```

Os métodos responsáveis pelo tráfego de dados são `dataSend` e `dataReceive`, cujos parâmetros são pacotes `packet_t` pertencentes ao namespace `PacketData` e formatados de acordo com seu tipo de dado e propósito. As primitivas usadas são `send` e `receive`, como é possível ver abaixo. Os pacotes de dados são formatados a partir do tipo de funcionalidade disponível que o criou, como explicado na seção 2.5.

```
ssize_t ServerConnectionManager::dataSend(int sfd, PacketData::packet_t
    packet) {
    auto bytes_sent = send(sfd, (void*) &packet, sizeof(PacketData::packet_t),
        0);

    return bytes_sent;
}

ssize_t ServerConnectionManager::dataReceive(int sfd, PacketData::packet_t&
    packet) {
    auto bytes_received = read(sfd, (void*) &packet,
        sizeof(PacketData::packet_t));

    return bytes_received;
}
```

2.6.2 SessionMonitor

`SessionMonitor` possui informações de todas sessões criadas por usuários logados através de uma estrutura `map` que relaciona o username de um dado perfil de usuário a seu `SessionController` correspondente. A classe verifica a existência de uma instância de `SessionController` do usuário em questão. Caso não exista, um objeto `SessionController` é criado para que, finalmente, seja chamado seu método de criação de sessão.

```
class SessionMonitor {
...
    std::map<std::string, SessionController*> _sessions;
...
};
```

2.6.3 SessionController

`SessionController` possui métodos de criação e encerramento de sessões. Um de seus atributos de classe é um `set` dos socket file descriptors correspondentes às conexões de usuários àquele perfil específico. Através desta classe é feito o gerenciamento de número de sessões existentes para um dado perfil de usuário, chamar métodos para seguir outros usuários e enviar e receber mensagens.

2.6.4 PersistenceManager

`PersistenceManager` pode ser acessada por todas sessões de cliente através de uma interface com um *PersistentUser*. A classe faz o gerenciamento do banco de dados e é capaz de realizar operações CRUD sobre dados de usuários.

2.6.5 MessageManager

Um usuário, ao enviar uma mensagem, a terá encaminhada pelo servidor ao `MessageManager`. Este é responsável por demultiplexar a mensagem para cada um dos seguidores do autor da mensagem. Isto é feito através de instâncias de `ProducerConsumerBuffer`s para cada seguidor. A mensagem é então adicionada de forma atômica ao buffer da instância caso exista espaço disponível. `ProducerConsumerBuffer` são alocados dinamicamente de acordo com a necessidade, i.e., só existem para um dado perfil de usuário se em algum momento da execução uma mensagem é direcionada a ele.

```
void MessageManager::processIncomingMessage(User& creator, const std::string
    text, const uint64_t timestamp) {
...
    _sem.wait();
    for (auto follower : creator.followers()) {
        if (!_pendingMessages.count(follower)) {
            _pendingMessages[follower] = new ProducerConsumerBuffer();
        }
    }
    _sem.notify();
...
    _sem.wait();
    for (auto follower : creator.followers()) {
        std::thread t = std::thread(&ProducerConsumerBuffer::enqueue,
            _pendingMessages[follower], incoming);
        threads.push_back(std::move(t));
```

```
    }
    _sem.notify();
...
};
```

O método para leitura de mensagens em buffer `getPacket` processa possíveis mensagens de maneira assíncrona e é limitada por um timeout de meio-segundo. Caso exista uma mensagem a ser lida, é retirada do buffer `ProducerConsumerBuffer` relacionado ao usuário que a recebeu e então retornada.

```
PacketData::packet_t MessageManager::getPacket(std::string toUser) {
...
    std::chrono::milliseconds timeout(500);
    std::future<message_t> future =
        std::async(&ProducerConsumerBuffer::dequeue, buffer);

    message_t message;
    message.packet.type = PacketData::packet_type::NOTHING;
    if (future.wait_for(timeout) == std::future_status::ready) {
        message = future.get();
    }

    return message.packet;
}
```

2.6.6 ProducerConsumerBuffer

A estrutura armazenada no buffer possue dados de `timestamp`, `packet` com a mensagem e outros dados relevantes, e um booleano `delivered` para controle de entrega.

```
typedef struct s_message {
    uint64_t timestamp;
    PacketData::packet_t packet;
    bool delivered;
} message_t;
```

A classe `ProducerConsumerBuffer` tem métodos discutidos anteriormente de manipulação de buffer `enqueue` e `dequeue`, responsáveis por adicionar ou remover mensagens de seu vetor. `ProducerConsumerBuffer` também possue índices para as operações de escrita e leitura (a fim de localizar o endereço do buffer a ser acessado) e de produções, cujo número informa a quantidade de mensagens armazenadas no buffer (a fim de bloquear a produção caso chegue em seu limite). Todos índices tem seu processo de escrita feito atomicamente para evitar problemas de consistência.

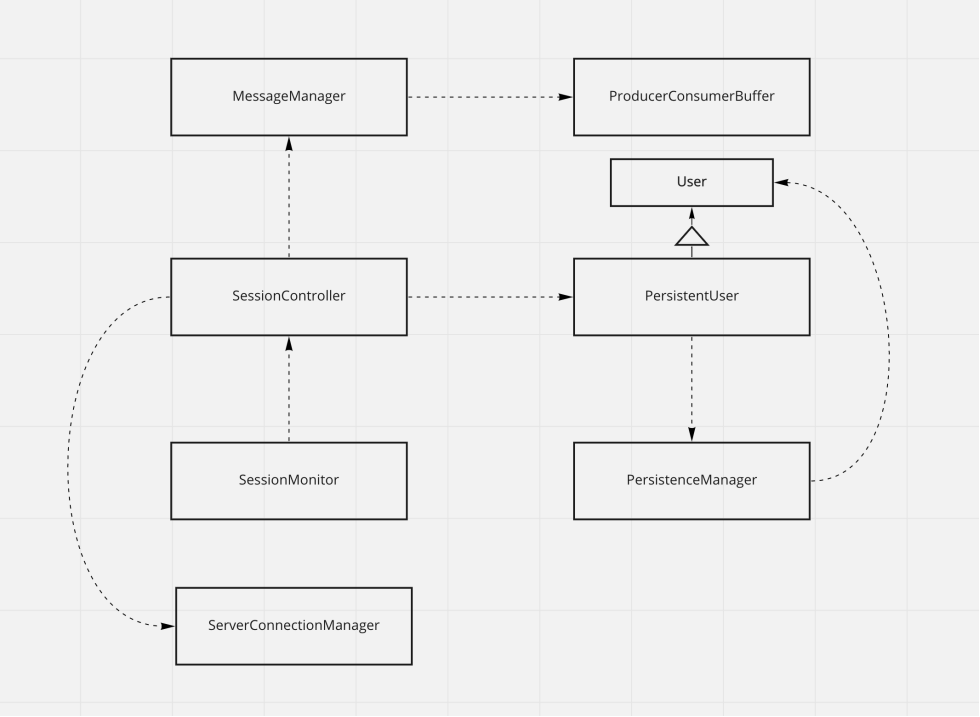

Figura 2.1: Diagrama de relacionamentos

3 Dificuldades e Desafios

Seguindo o roteiro já existente do trabalho, e após definir um roadmap inicial, a sua implementação foi relativamente direta mesmo com alguns de nossos integrantes não tendo familiaridade com C++. Nos reunimos online periodicamente para discutir o trabalho e tirar dúvidas.
A principal dificuldade identificada pelo grupo durante o desenvolvimento do trabalho foi em relação a testagem da concorrência e estados internos do programa, para a qual não foi encontrada uma solução definitiva. Em especial, com referência a testes automatizados. Até por isso, foi difícil ganhar confiança de que conseguimos garantir que as mensagens são entregues em ordem cronológica quando as timestamps são extremamente próximas e as mensagens são escritas no ProducerConsumerBuffer e consumidas na ordem de entrada.
Também tivemos dificuldades em manter a independência entre os módulos, tendo ainda em nossa implementação módulos interdependentes em que o fluxo de controle depende

do comportamento exterior ao módulo. Outras problemas costumeiramente encontrados foram os deadlocks. Neste último caso precisamos interromper o processo manualmente, i.e., dependemos de outro processo para encerrar o nosso processo. Outro problema um tanto quanto laborioso foi a captura bloqueante do input de usuário, resolvido através de chamadas assíncronas com timeout.

3.1 Vídeo

O vídeo para demonstração das funcionalidades básicas se encontra em `https://www.youtube.com/watch?v=_u9_rbYv2LQ`.