

Universidade Federal do Rio Grande do Sul

Instituto de Informática - Departamento de Informática Aplicada



Sistemas Operacionais II N - Turma A
Professor Weverton Cordeiro

Trabalho Prático 01

Um chat cliente servidor utilizando sockets

Douglas Gehring - 243682
Henrique Valcanaia - 240501
Leonardo Felipe Mendes - 250383
Rodrigo Cardoso Buske - 206526

Porto Alegre, 11 de Outubro de 2020.

Especificação do Trabalho

A proposta do projeto é implementar um serviço de troca de mensagens instantâneas, para permitir a comunicação sincronizada entre vários usuários em grupos. A proposta deve ser desenvolvida em duas etapas. A primeira etapa compreende funcionalidades que dependerão de tópicos como threads, processos, comunicação e sincronização para serem implementadas. A aplicação deverá executar em ambientes Unix (Linux), mesmo que tenha sido desenvolvida em outras plataformas. O projeto deverá ser implementado em C/C++, usando a API Transmission Control Protocol (TCP) sockets do Unix.

Introdução/Organização do Projeto

Durante as fases iniciais fizemos reuniões em dias alternados para leitura em conjunto da especificação, para analisar se todos estavam com a mesma compreensão perante o escopo do projeto bem como as funcionalidades necessárias a serem implementadas.

Após o entendimento, começamos a desenhar um esqueleto em forma de diagrama, para que pudéssemos visualizar o que seria essencial para questões de organização e estrutura inicial do projeto. Para isso, utilizamos o draw.io. Segue abaixo o nosso esboço inicial (que está um pouco desatualizado em relação organização final):

```
typedef struct __packet{
    uint16_t type;
    uint16_t seqn;
    uint16_t length;
    uint16_t timestamp;
    const char* _payload;
} packet;

typedef struct _message{
    uint16_t timestamp;
    const char* group;
    const char* username;
    const char* text;
} message;

typedef struct _connection{
    char* ip;
    char* port;
} connection;

typedef struct _userInfo{
    char* name;
    char* group;
} userInfo;
```

Figura 1: structs iniciais do projeto

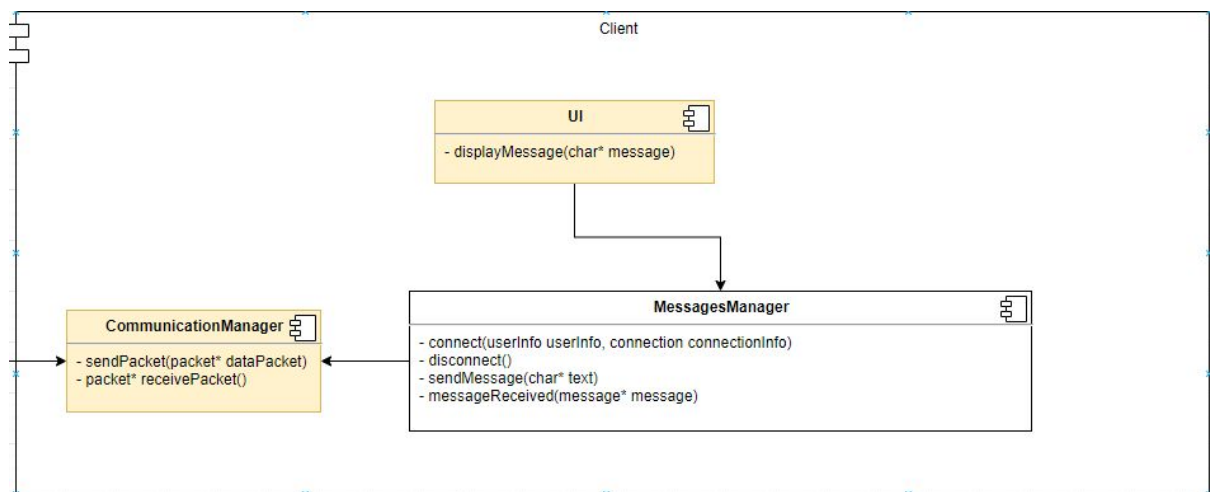


Figura 2: diagrama inicial do client

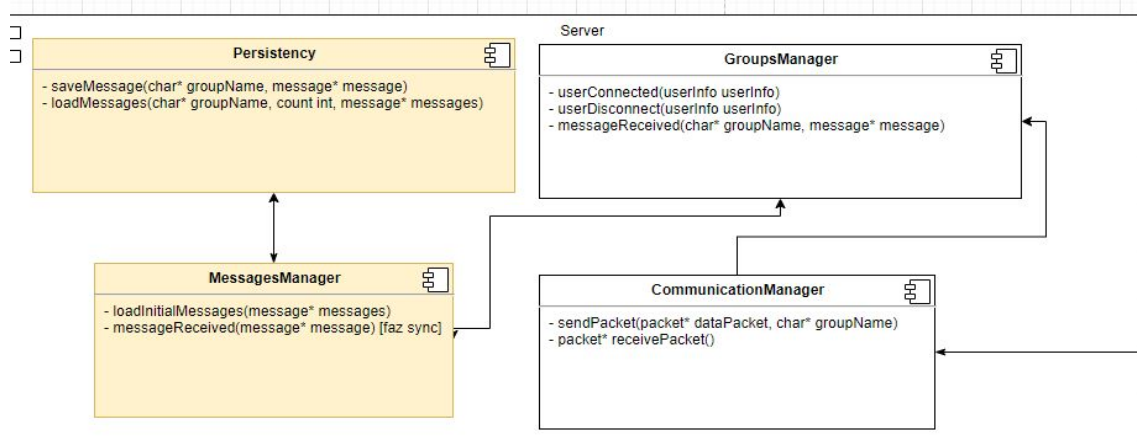


Figura 3: diagrama inicial do server

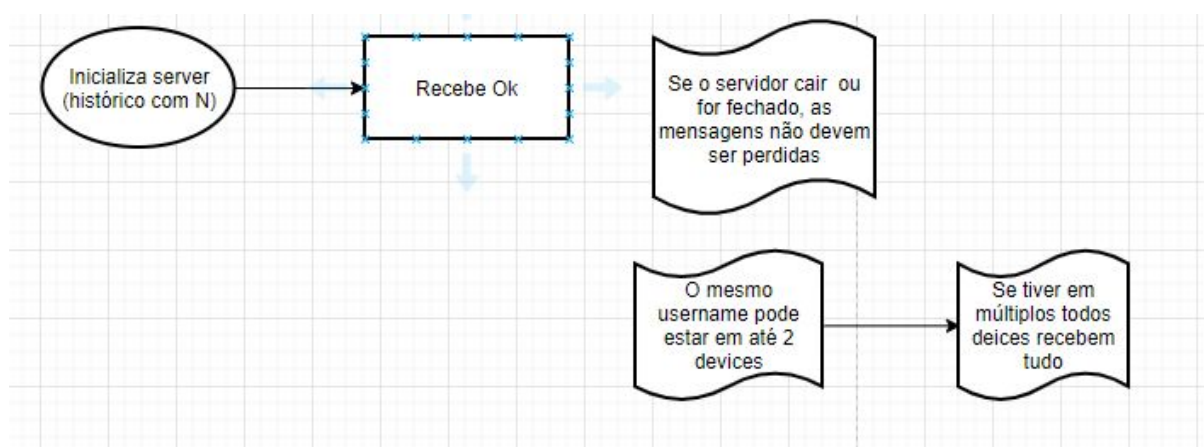


Figura 4: Fluxo de Tarefas inicial do Server

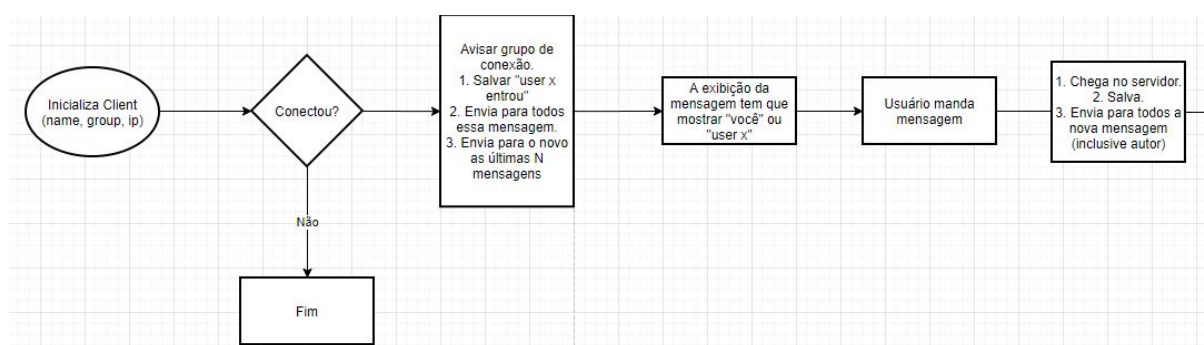


Figura 5: Fluxo de Tarefas inicial do Cliente parte 1

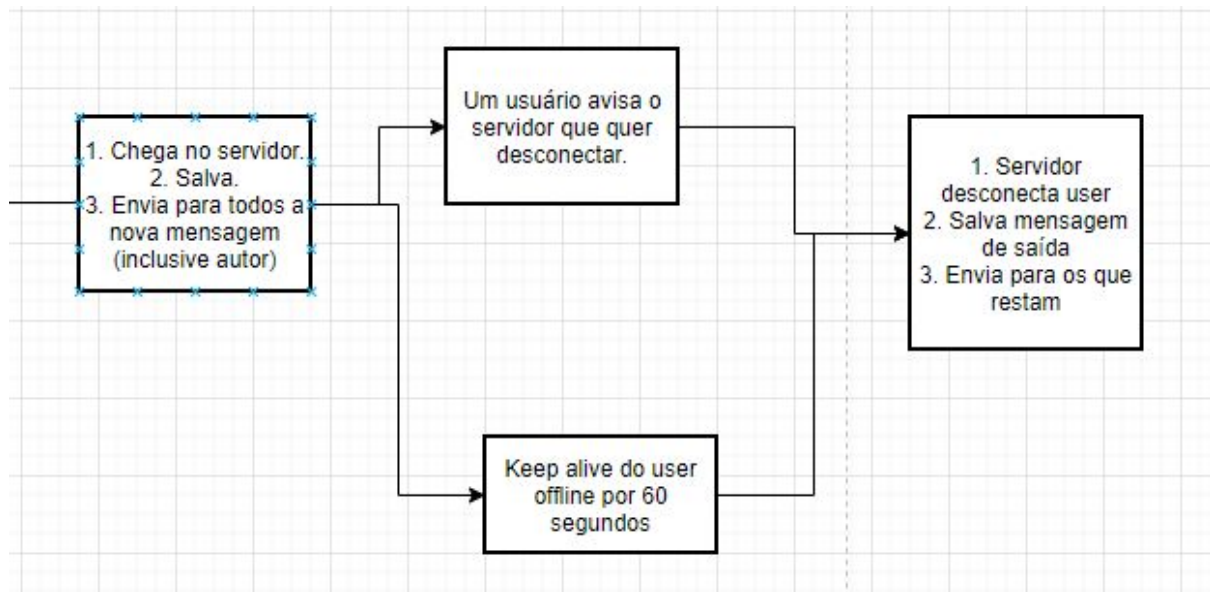


Figura 6: Fluxo de Tarefas inicial do Cliente parte 2

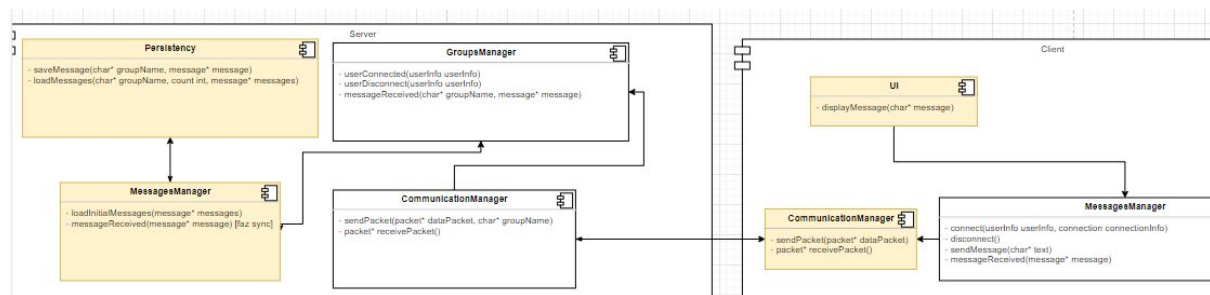


Figura 7: diagrama completo server e client

A organização do projeto foi o primeiro passo do trabalho, separamos o projeto em um biblioteca compartilhada entre *server* e *client* chamada de *shared_lib*, e uma biblioteca e executável para cada, *server_lib*, *server*, *client_lib*, *client*.

A divisão dos produtos entre executável e biblioteca foi feita para permitir utilizar testes automatizados para o código desenvolvido, mantendo o executável apenas com o *main.cpp* com quase nenhuma lógica. Adicionamos a biblioteca do [googletest](#) da Google para utilizarmos testes unitários, porém acabamos por não implementar nenhum teste devido a falta de tempo.

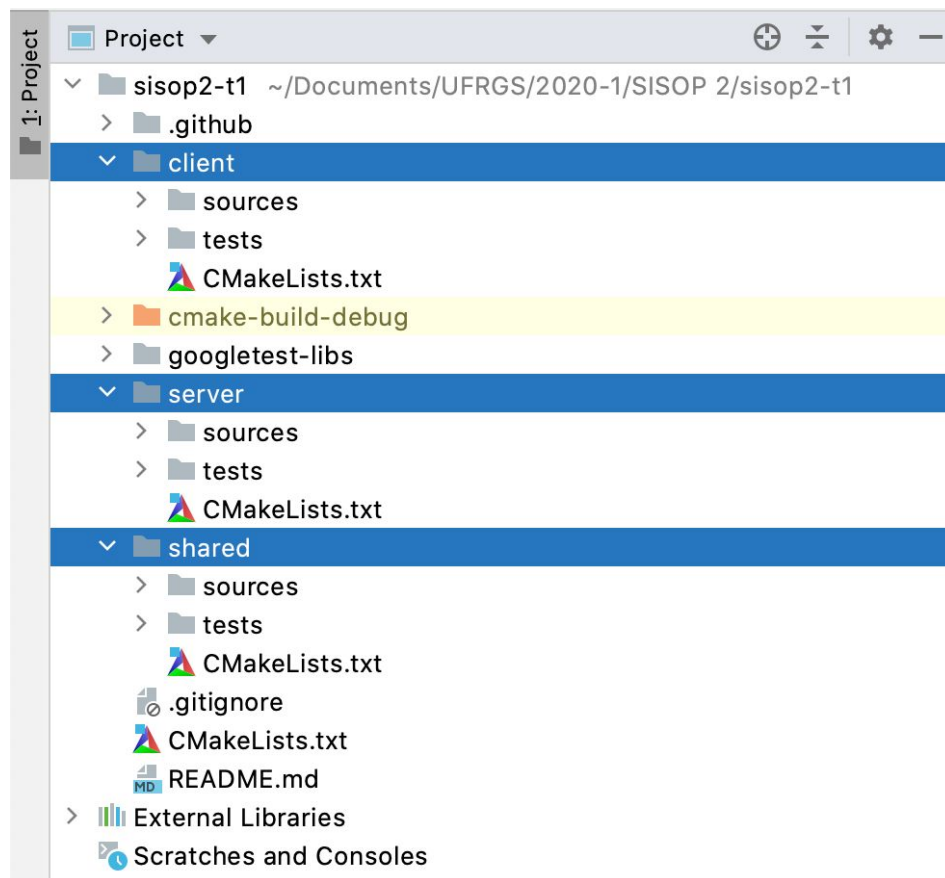


Figura 8: Separação server, client e shared

Descrição dos Ambientes de desenvolvimento e testes

No grupo foram utilizados dois sistemas operacionais distintos, a maior parte do tempo foi macOS 10.15.7, mas também foi utilizando o Ubuntu numa VM no Windows. Importante salientar que desenvolvemos e testamos tanto no macOS quanto no Ubuntu, incluindo comunicação entre os sistemas distintos.

Como demoramos um pouco para fazer os primeiros testes com Ubuntu acabamos encontrando um pouco tarde alguns problemas, um relacionado a serialização, que é melhor descrito na seção sobre dificuldades, e outro frequente era por *includes* adicionarem coisas um pouco diferentes em cada sistema, as vezes necessitando mudar o *include* utilizado.

Todos os membros da equipe utilizaram a IDE CLion da JetBrains, a qual é uma excelente IDE para projetos que são implementados nas linguagens C/C++ (as instruções para rodar o projeto não necessitam do CLion).

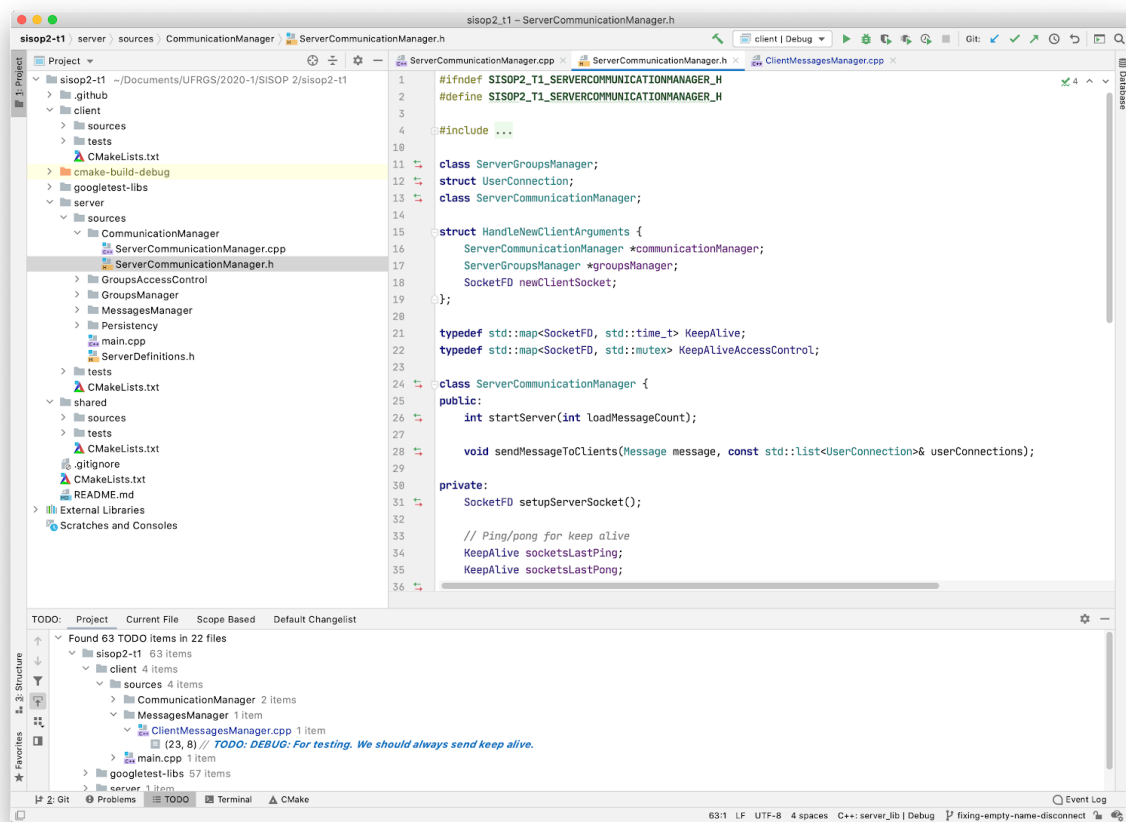


Figura 9: Interface da IDE CLion

Para controle de versões, utilizamos o Git na plataforma GitHub. Também utilizamos o Tower, um cliente de Git para especialmente facilitar visualização de alterações e commit de linhas individuais.

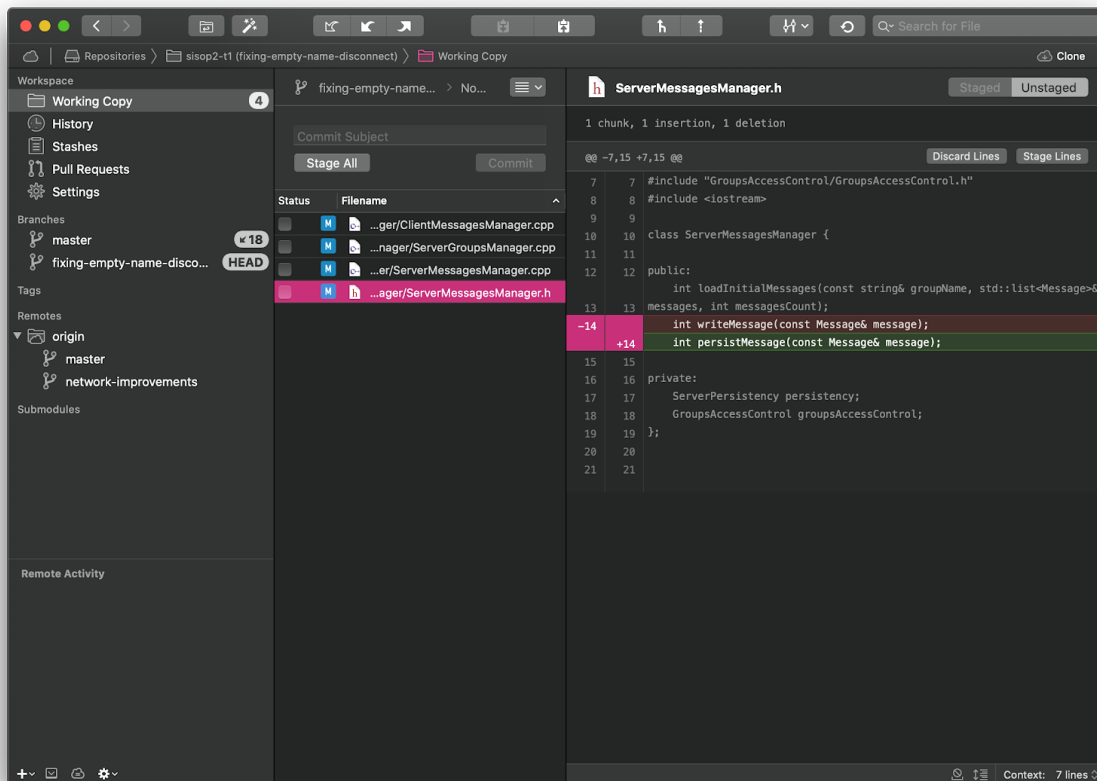


Figura 10: Interface do Tower

O GitHub também foi bastante importante para garantir o funcionamento entre os sistemas operacionais. Utilizamos o [Github Actions](#) para compilar o projeto no macOS e Ubuntu a cada "pull request", evitando problemas de integração. O projeto pode ser encontrado em <https://github.com/Robuske/sisop2-t1>.

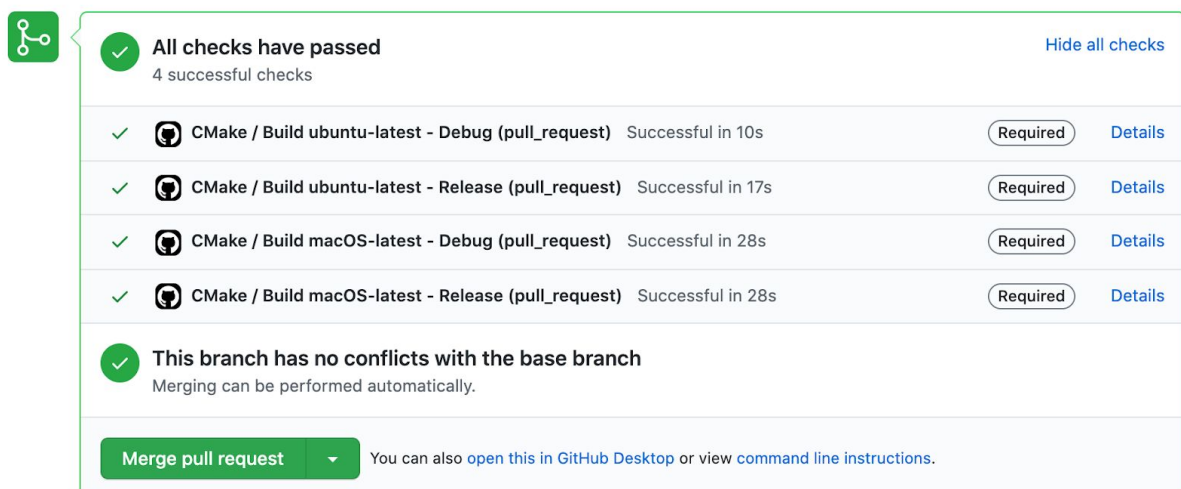


Figura 11: Checklist para as releases tanto em ambientes Linux x macOS

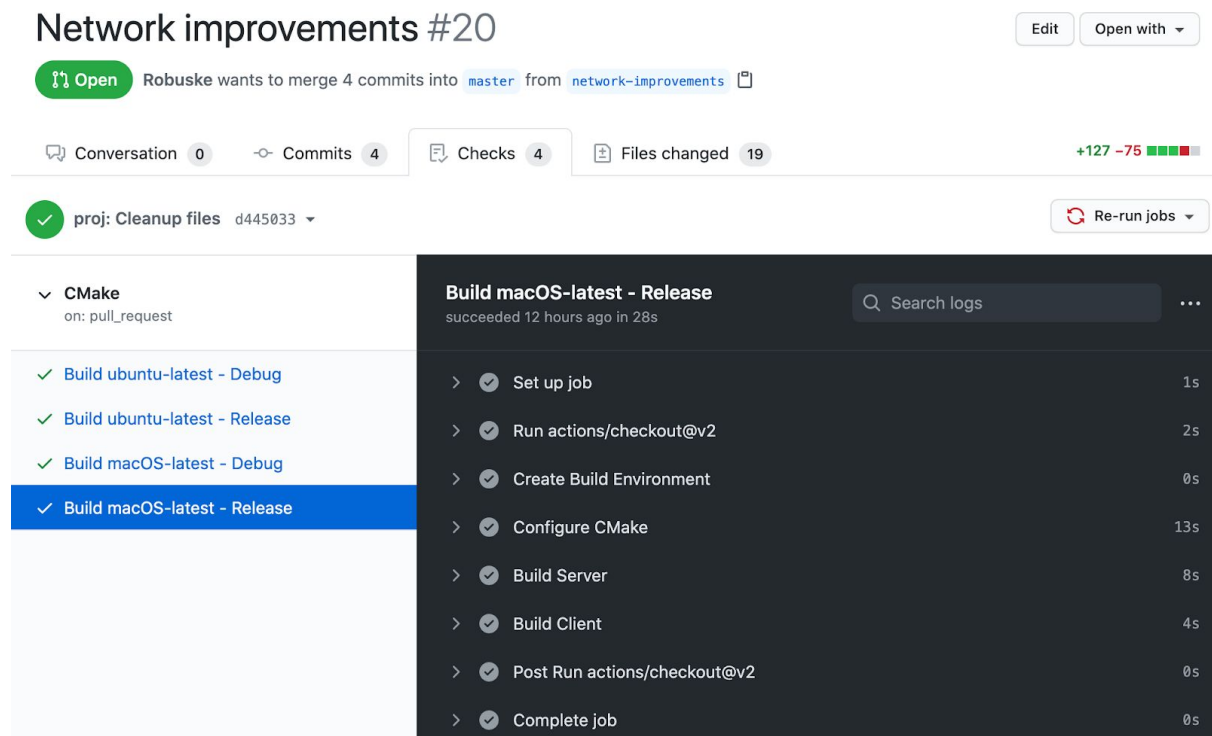


Figura 12: Detalhe do tempo de build para macOS

O ambiente utilizado para testes foi uma máquina com o sistema operacional Ubuntu Linux 20.04 LTS, com 8 GB de Memória RAM, processador Intel Core i5 8th Gen 1.8 Ghz.

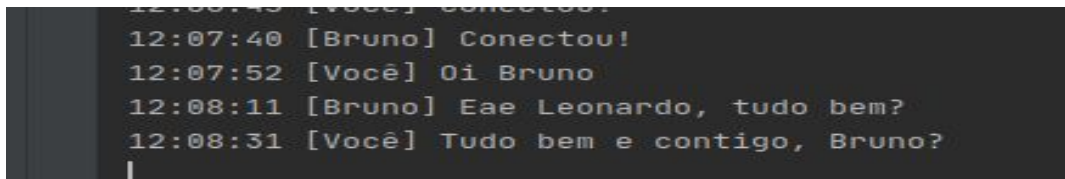
Para compilar o projeto, foram necessários alguns passos iniciais listados abaixo. Atenção para o fato de que a versão utilizada do C++ é a 14, e a do CMake é 3.17, o passo 4 deve baixar a versão mais atual do CMake. Não ficou muito claro quando é necessário o passo 5, mas parece ser relacionado a ter ocorrido mudanças na estrutura do projeto (como trocar de *branch*), ao rodar a primeira vez provavelmente não vai precisar:

Pré-requisitos necessários para a “build” e “run” do projeto (Linux):

1. Abrir o terminal do Ubuntu
2. `sudo apt-get update`
3. `sudo apt-get install build-essential`
4. (opcional) `sudo snap install cmake --classic`
5. (opcional) `cmake --configure .`
6. `cmake --build . --target server`
7. `cmake --build . --target client`
8. Os executáveis estarão disponíveis em `server/server` e `client/client`

Observamos durante o *troubleshooting* que o Ubuntu não estava completamente atualizado, e através da execução desses comandos a compilação em C++ e C foi possível, visto que durante o passo 3 é instalado tanto o compilador GCC quanto o Cpp.

Segue abaixo um exemplo do software do projeto em execução:



```
12:07:40 [Bruno] Conectou!  
12:07:52 [Você] Oi Bruno  
12:08:11 [Bruno] Eae Leonardo, tudo bem?  
12:08:31 [Você] Tudo bem e contigo, Bruno?
```

Figura 13: Conversa entre Leonardo (Você) e Bruno.

Processos de Comunicação, gerência de threads e sincronização do Projeto

Como foi implementada a concorrência no servidor para atender múltiplos clientes

A concorrência no servidor para atender múltiplos clientes foi implementada utilizando duas threads no *server* para cada nova conexão de um cliente. Uma thread é responsável pela escrita/leitura de dados nos sockets, enquanto a outra é utilizada para realizarmos o controle do keep-alive.

Primeiramente inicializamos as configurações de socket no servidor e esperamos por uma conexão de um novo cliente. Quando uma conexão é estabelecida criamos uma thread que terá as seguintes responsabilidades:

- Receber as mensagens enviadas pelo cliente;
- Writeback das mensagens recebidas;
- Verificação do número máximo de conexões por nome de usuário (2);
- Tratamento de erros de conexão.

As threads responsáveis por escrita/leitura cada conexão acessam os seguintes recursos compartilhados no servidor:

- Lista geral de grupos;
- Lista de usuários conectados por grupo;
- Arquivo, de cada grupo, responsável pela persistência das mensagens.

Como temos múltiplas threads acessando recursos compartilhados, precisamos implementar técnicas de exclusão mútua para garantir sincronia no acesso a dados. A técnica utilizada para realizar a sincronia foi a utilização de um mutex por grupo e um mutex geral para todos os grupos, responsável pelo controle ao acesso à lista geral de grupos.

A thread de **keep-alive** é responsável por verificar se o cliente está responsivo. A cada *TIMEOUT* segundos, que pela especificação do trabalho é 60, verificamos se o tempo entre o **último ping** (uma mensagem especial enviada pelo servidor) e o **último pong** (atualizado por qualquer mensagem enviada pelo cliente) é menor que *TIMEOUT* segundos. Caso o tempo entre o **ping** e o **pong** for menor que 60 segundos enviamos um novo ping e reiniciamos a espera, caso contrário desconectamos o cliente do servidor.

Em quais áreas do código foi necessário garantir a sincronização no acesso a dados

Para garantir a consistência e sincronização de dados utilizamos *mutex* e encapsulamos o controle de acesso aos recursos compartilhados pelos grupos a uma classe chamada **GroupsAccessControl**, cuja definição é dada abaixo:

```
class GroupsAccessControl {  
private:  
    SharedResourcesAccessControl accessControl  
public:  
    void lockAccessToGroup(const string& groupName);  
    void lockAccessToGroup(const string& groupName);  
};
```

O tipo `SharedResourcesAccessControl` é definido pela seguinte declaração:

```
typedef std::map<string , std::mutex> SharedResourcesAccessControl;
```

Na estrutura `std::map<string , std::mutex>` temos que cada grupo é uma *key* e o mutex relacionado a cada grupo é um *value*, portanto o acesso e utilização do mutex de cada grupo é dado por :

Primitivas de controle de um mutex	Métodos de acesso para cada grupo
Primitiva lock do mutex	<code>this->accessControl[groupName].lock();</code>
Primitiva unlock do mutex	<code>this->accessControl[groupName].unlock();</code>

Utilizamos os métodos lock e unlock para controlar o acesso de cada grupo aos recursos compartilhados presentes no projeto. Quando algum usuário acessa um recurso compartilhado pelo seu grupo precisamos garantir que as operações de escrita e leitura sejam consistentes e não apresentem divergências. Um exemplo de inconsistência nas operações de escrita e leitura é o caso de dois usuários escreverem simultaneamente no arquivo relativo ao grupo ou até mesmo uma operação de escrita começar enquanto uma operação de leitura ainda não terminou. Segue uma descrição dos recursos compartilhados.

Recursos compartilhados	Utilizadores
Lista geral de grupos	Usuários de qualquer grupo.
Lista de usuários conectados por grupo	Usuários conectados ao mesmo grupo.
Arquivo de cada	Usuários conectados

grupo	ao mesmo grupo.
-------	-----------------

Teoricamente, as operações de leitura dos recursos compartilhados podem ser realizadas por mais de um usuário ao mesmo tempo, no entanto, as operações de escrita precisam ser bloqueantes para garantir que as operações de leitura possuam dados consistentes. Na implementação feita pelo grupo optamos por uma maneira mais simplificada de controle, ou seja, tanto a leitura quanto a escrita são operações bloqueantes.

A lista geral de grupos que contém todos os grupos criados e todos os usuários conectados a esses grupos é tratada de uma maneira especial, já que seu acesso é bloqueado a partir do momento que um usuário de **qualquer grupo** faz um acesso para leitura/escrita.

Descrição das principais estruturas e funções que a equipe implementou

SocketConnectionInfo

Estrutura utilizada no *client* para identificar o endereço e porta do servidor que estamos nos conectando.

```
struct SocketConnectionInfo {  
    string ipAddress;  
    unsigned short port;  
};
```

SocketFD

Alias criado para adicionar valor semântico aos descritores de arquivo dos sockets.

```
typedef int SocketFD;
```

ClientCommunicationManager

Responsável pelo envio e recebimento de Packets entre cliente e servidor. Note que o Packet não é exposto na API pública - camadas mais altas do app conhecerão apenas Message.

ClientMessagesManager

Classe responsável por ler e escrever mensagens. Esta classe possui 2 threads - uma para ler e outra para escrever mensagens. Ficamos em dúvida sobre onde especificar a thread de read, mas achamos que fazia mais sentido atribuir para *ClientMessageManager*, uma vez que ela terá de ser responsável pela sua leitura e escrita no *client*.

ClientUI

Classe responsável pelo controle da UI. Contém métodos de formatação de dados, exibição de mensagens e erro.

ServerCommunicationManager

É a classe principal do servidor.

Esta classe irá iniciar 2 threads a cada conexão - uma para o mecanismo de keep alive e outra para leitura do socket da nova conexão.

Além disso, ela possui algumas funções auxiliares para funções como ler um Packet do socket, escrever uma Message para uma lista de *UserConnection*, terminar a conexão com o cliente, fechar o socket, e atualizar último ping/pong com controle de exclusão mútua, por exemplo.

GroupsAccessControl

Wrapper que abstrai e esconde a lógica para travar e liberar os mutexes de leitura e escrita na persistência do grupo e no acesso às listas compartilhadas que contém as informações de todos os grupos. Esta estrutura é utilizada nas classes *ServerGroupsManager* e nas classes *ServerMessagesManager*.

ServerGroupsManager

Classe responsável por gerenciar os grupos, enviar mensagem, enviar mensagens para um usuário específico, carregar as N mensagens iniciais de um grupo, lidar com conexão e desconexão, controlar o número de conexões simultâneas de um mesmo usuário e buscar o nome de um usuário a partir do socket.

ServerMessagesManager

Esta classe é responsável por adicionar o controle de exclusão mútua ao se comunicar com a persistência para carregar as últimas N mensagens e persistir as mensagens.

Packet

Estrutura utilizada para envio e recebimento de dados entre servidor e cliente. Inicialmente, utilizamos uma struct com string que funcionavam no macOS, porém, por problemas de serialização, tivemos que utilizar uma lista de char de tamanho fixo.

Message

Classe criada para ser utilizada nos níveis mais altos de abstração no servidor e cliente, por sua maior facilidade de utilização devido as string. É uma versão melhorada do Packet.

UserInfo

Estrutura utilizada no cliente para encapsular o nome do usuário e do grupo.

```
struct tUserInfo {  
    string username;  
    string groupName;  
} typedef UserInfo;
```

PacketType

Enum criado para identificar o tipo do pacote.

```
enum PacketType {  
    TypeConnection,  
    TypeDisconnection,  
    TypeDesconnection,
```

```
TypeMessage,  
TypeKeepAlive,  
TypeMaxConnectionsReached  
};
```

Explicar o uso das diferentes primitivas de comunicação;

Detalhes sobre a utilização já foram indicados anteriormente. Utilizamos as primitivas `read(int fd, void *buf, size_t count)` e `write(int fd, const void *buf, size_t count)` para ler e escrever no socket.

Como utilizamos threads e não processos, não precisamos utilizar a comunicação entre processos.

Problemas e dificuldades encontradas durante a implementação do trabalho

O maior problema foi lidar com as idiossincrasias do C++. O grupo no geral teve dificuldade em lidar com as particularidades da linguagem, uma vez que C/C++ não é uma linguagem presente no dia a dia dos integrantes. Talvez se utilizássemos uma linguagem de mais alto nível em que tivéssemos primitivas equivalentes, poderíamos focar muito mais no conteúdo do trabalho em si.

Um item em especial que chamou nossa atenção foi a necessidade de criação de um método estático para utilizar a referência na criação da thread.

Outro item importante, foi entender as diferenças na forma como os pacotes eram enviados/recebidos em sistemas operacionais diferentes. No início estávamos enviando uma struct com campos do tipo *string*, e isso funcionava perfeitamente no macOS, porém, ao testar no Ubuntu, ocorria *segmentation fault*. Nossa suspeita era relacionada a serialização e desserialização dos dados, o que foi confirmado ao converter *strings* para *char*, ou seja, agora estávamos enviando simplesmente um *array* de bytes. Perdemos uma quantidade considerável de tempo para alterar isso.

Uma sugestão para os próximos semestres é a criação de uma base de conhecimentos, estilo FAQ.

Lendo a respeito de outros problemas também percebemos que a leitura do *buffer* poderia não trazer todos *bytes* pedidos, ou seja, mesmo que a gente sempre envie um pacote de 500 *bytes*, pode ser que a primeira leitura retorne 100 e a segunda 400, ou ainda existir dois pacotes disponíveis, uma leitura trazer 200 e a seguinte 800. Para evitar esse problema mantemos um *buffer contínuo* com os *bytes* ainda não consumidos pela leitura, e montamos o pacote quando temos *bytes* suficientes.

No fim, como esse é um problema muito dependente de fatores externos não foi possível reproduzir para efetivamente testar se a solução funcionou, mas a lógica do que está sendo feito parece fazer sentido.

Tivemos bastante dúvida de como estruturar quem conhecia quem e onde eram criadas as threads. A solução que acabamos escolhendo, aonde o "controlador" do *server* é o *ServerCommunicationsManager* e o do *client* é o *ClientMessagesManager*, foi baseada no fato de que no *server* o fluxo de dados só vem a partir do *ServerCommunicationManager*, enquanto no *client* pode vir do usuário digitando, ou recebendo informações do *ClientCommunicationManager*.