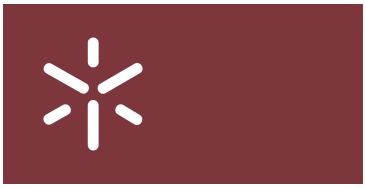

Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Unidade Curricular de Segurança de Sistemas Informáticos

Ano Letivo de 2023/2024

Serviço Local de Troca de Mensagens

David Teixeira A100554
João Pedro Pastore A100543
Luís Ferreira A91672

Maio, 2024

Data da Receção	
Responsável	
Avaliação	
Observações	

Serviço Local de Troca de Mensagens

David Teixeira A100554

João Pedro Pastore A100543

Luís Ferreira A91672

Maio, 2024

Índice

1. Introdução **2**

2. Arquitetura Funcional **3**

3. Reflexão **4**

3.1. **Funcionalidades do Serviço** 4

3.1.1. Enviar uma Mensagem de Texto para um Destinatário. 4

3.1.2. Listar as Novas Mensagens de um Utilizador 5

3.1.3. Ler o Conteúdo de uma Mensagem 5

3.1.4. Responder ao Remetente de uma Mensagem Previamente Recebida 6

3.1.5. Apagar uma Mensagem Recebida 6

3.1.6. Criar um Grupo de Utilizadores do Serviço 6

3.1.7. Remover um Grupo de Utilizadores do Serviço 7

3.1.8. Listar os Utilizadores Membros de um Grupo 7

3.1.9. Adicionar um Utilizador a um Grupo 7

3.1.10. Remover um Utilizador de um Grupo 7

3.2. Segurança do Serviço 8

3.2.1. Proprietários e Permissões dos Ficheiros do Sistema 8

3.2.2. Proprietários e Permissões dos Processos 8

3.3. **Fundamentação da Arquitetura e Desenvolvimento do Sistema de Comunicação** 9

4. Conclusão **10**

5. Anexos **12**

5.1. `Daemon.c` 12

5.2. `Client.c` 15

5.3. `Group.c` 19

5.4. `Message.c` 24

5.5. Imagens 32

1. Introdução

Este relatório descreve a solução proposta para o desenvolvimento de um sistema de mensagens interativo em **C**, que consiste num *daemon* e num cliente para troca de mensagens entre utilizadores e grupos. O sistema foi desenvolvido com o objetivo de fornecer uma plataforma de comunicação eficiente e segura para utilizadores, permitindo o envio, receção e gestão de mensagens individuais e de grupos.

O sistema é composto por dois programas principais: o *daemon*, responsável por receber, processar e armazenar as mensagens, e o cliente, que permite aos utilizadores interagirem com o *daemon*, enviando e recebendo mensagens, criando grupos, entre outras funcionalidades. O *daemon* utiliza *named pipes* (**FIFO's**) para comunicação com os clientes, enquanto que o cliente comunica com o *daemon* através de *system calls* de leitura e escrita em *pipes*.

As principais preocupações durante o desenvolvimento do sistema foram criar um ambiente seguro (através da utilização das ferramentas do sistema operativo) e a facilidade de uso para os utilizadores. O objetivo era criar uma plataforma capaz de lidar com um grande número de utilizadores, garantindo ao mesmo tempo a integridade e confidencialidade das informações trocadas.

Uma das decisões arquiteturais mais relevantes foi a escolha do modelo de comunicação baseado em *named pipes* para a troca de mensagens entre o *daemon* e os clientes. Esta escolha proporciona uma forma eficiente de comunicação bidirecional entre os processos, garantindo baixa latência e alto desempenho na entrega das mensagens. Adicionalmente, a utilização de *named pipes* simplifica a implementação e integração do sistema, pois não requer a configuração de um *daemon* de comunicação separado.

2. Arquitetura Funcional

O sistema é composto por dois programas principais: o *daemon* (`daemon.c`) e o cliente (`client.c`), e por programas secundários como `message.c` e `group.c`, responsáveis por funcionalidades relativas a mensagens e grupos, respetivamente.

O *daemon* é responsável por receber e processar as mensagens enviadas pelos clientes, armazenando-as em ficheiros no *file system*, e o cliente permite aos utilizadores interagirem com o *daemon*, enviando mensagens, criando grupos, entre outras operações.

O sistema utiliza *pipes* com nome para a comunicação entre o *daemon* e os clientes, que fornecem uma interface de comunicação eficiente entre os processos, permitindo a troca de mensagens de forma assíncrona. Além disso, o sistema utiliza ficheiros no *file system* para armazenar as mensagens enviadas pelos utilizadores, garantindo persistência e recuperação das informações.

O programa utiliza uma estrutura de dados para representar uma mensagem ou um pedido. A `struct Message`, contém campos como **ID** (`int id`), tipo (`char type[50]`), remetente (`char sender[50]`), destinatário (`char receiver[50] ou char group[50]`) e conteúdo (`char content[512]`):

As duas principais componentes de *software* desenvolvidas são executadas como processos independentes. O *daemon* depende das bibliotecas padrão de **C** (utilizando funções como `system()`) e de algumas *system calls* para comunicação e manipulação de ficheiros, enquanto que o cliente também depende das bibliotecas padrão de **C** e de *system calls* para comunicação com o *daemon*. Ambas as componentes dependem das funções existentes nos programas secundários, `message.c` e `group.c`, sendo que o cliente depende apenas do programa secundário relativo a mensagens.

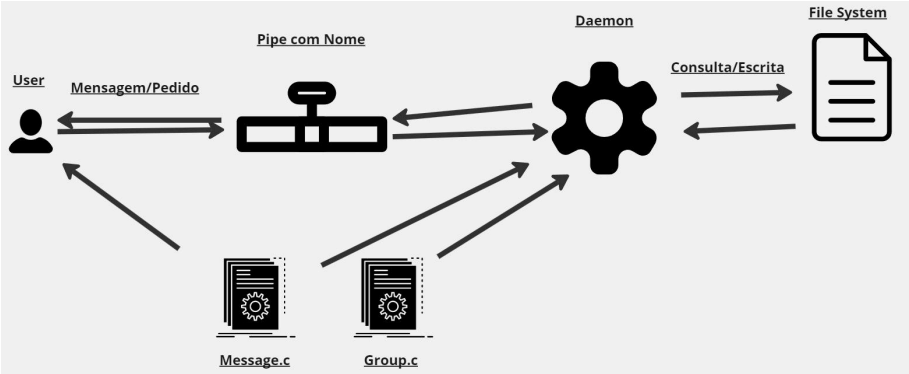

Figura 1: Arquitetura Funcional[1]

Os comandos disponíveis no serviço incluem a criação de grupos, envio de mensagens individuais e para grupos, listagem de mensagens recebidas, entre outras operações. A secção seguinte irá abordar as funcionalidades implementadas no serviço.

[1]O diagrama não reflete por completo a realidade da funcionalidade do programa, pois o *daemon* retorna um *pointer* para o ficheiro que o utilizador deseja ler, em vez de o enviar via *pipe*.

3. Reflexão

3.1. Funcionalidades do Serviço

As funcionalidades do serviço podem ser separadas em duas categorias :

- Funcionalidades relativas a grupos
- Funcionalidades relativas a mensagens e utilizadores

De forma a cumprir o aspeto de modularidade e encapsulamento das componentes de *software* desenvolvidas, foi necessária a criação de dois ficheiros : *message.c* e *group.c*, responsáveis pela gestão das funcionalidades seguidamente descritas:

3.1.1. Enviar uma Mensagem de Texto para um Destinatário.

A função `handle_write_command_` (Figura 6) é responsável por escrever uma mensagem num ficheiro específico, destinado a um utilizador ou grupo.

Primeiro, constrói o *path* do ficheiro onde a mensagem será armazenada utilizando `sprintf`. Este *path* é baseado na diretoria `MESSAGE_FOLDER`, no nome do grupo, no destinatário da mensagem e num contador.

Seguidamente, são definidas permissões de acesso ao ficheiro utilizando `chmod`, bloqueando a leitura e escrita para todos.

Obtém-se o **ID** do utilizador e do grupo do destinatário da mensagem utilizando `getpwnam` e `getgrnam`, respectivamente. Se o utilizador ou o grupo não existirem, são definidos como "*no user*" e "*no group*".

Depois, é verificado se a diretoria de mensagens existe, e se não, é criada.

Então, o ficheiro é aberto em modo de escrita. Se ocorrer algum erro na abertura do ficheiro, uma mensagem de erro é impressa e a função retorna.

A mensagem é escrita no ficheiro junto com o remetente e o conteúdo.

Após escrever a mensagem no ficheiro, é definido o seu dono e o seu grupo usando `chown`, e são ajustadas as permissões de leitura e escrita para o proprietário e grupo do ficheiro, respectivamente, utilizando `chmod`.

```
// Mudar permissões para dono
if (pwd != NULL)
  chmod(file_path, 0400);

// Mudar permissões para grupo
if (grp != NULL)
  chmod(file_path, 0440);
```

Finalmente, uma mensagem de sucesso é enviada para o *file descriptor* do *daemon* indicando que a mensagem foi enviada com sucesso. O contador de mensagens é incrementado para garantir que os nomes dos ficheiros sejam únicos.

3.1.2. Listar as Novas Mensagens de um Utilizador

A função `handle_list_command` (Figura 8) realiza a listagem das mensagens disponíveis para um determinado remetente. Primeiramente, abre a diretoria onde as mensagens são armazenadas. Se ocorrer algum erro durante essa operação, uma mensagem de erro é enviada ao *daemon*.

De seguida, a função itera sobre cada ficheiro na diretoria. Para cada ficheiro, verifica se o nome corresponde ao remetente da mensagem atual. Se corresponder, extrai o nome do remetente e o **ID** da mensagem do nome do ficheiro.

Depois, constrói uma resposta contendo o remetente e o **ID** de cada mensagem encontrada. Esta resposta é enviada ao *daemon* para ser transmitida ao cliente.

Finalmente, após iterar sobre todos os ficheiros, a diretoria é fechada. Esta função é essencial para permitir que o cliente saiba quais mensagens estão disponíveis para leitura.

Se a listagem for invocada com o argumento **-a**, será impressa a totalidade das respostas recebidas. Esta funcionalidade foi implementada com o auxílio da função `handle_listall_command`.

Esta função, incialmente, abre a diretoria onde as mensagens estão armazenadas utilizando a função `opendir()`. Se a diretoria não puder ser aberta, imprime uma mensagem de erro e retorna.

Seguidamente, inicializa uma *string* chamada *response* para armazenar a resposta que será enviada ao cliente.

Depois, itera sobre cada entrada na diretoria utilizando `readdir()` até que não haja mais entradas. Para cada entrada, verifica se o nome do ficheiro contém o nome do remetente da mensagem atual.

Se o nome do ficheiro corresponder ao remetente, extrai o nome do remetente do ficheiro, abre o ficheiro correspondente e lê o nome do remetente até o caractere $ utilizando `fscanf()`. De seguida, remove a parte do nome do ficheiro que indica o grupo de destinatários utilizando `strtok_r()`.

Depois, tokeniza o nome do ficheiro utilizando _ como delimitador e obtém o segundo *token*, que representa o **ID** da mensagem. Converte esse *token* num número inteiro utilizando `atoi()`.

Finalmente, adiciona o remetente e o **ID** ao final da *string response* que será enviada ao cliente.

Após listar todas as mensagens do remetente, fecha a diretoria utilizando `closedir()` e envia a resposta ao cliente utilizando `write()`.

3.1.3. Ler o Conteúdo de uma Mensagem

A função `handle_read_command` (Figura 7) procura e envia o *path* do ficheiro associado a uma mensagem específica, identificada pelo seu **ID**. Inicialmente, abre a diretoria onde as mensagens estão armazenadas. Se houver um erro durante a operação, uma mensagem de erro é enviada ao *daemon*.

De seguida, a função itera sobre cada ficheiro na diretoria de mensagens, verificando se o nome do ficheiro corresponde ao padrão esperado para ficheiros de mensagem e extraindo o **ID** da mensagem do nome do ficheiro.

Após extrair o **ID** do ficheiro, verifica se corresponde ao **ID** especificado na mensagem recebida. Se encontrar um ficheiro com o **ID** correspondente, define uma variável como verdadeira e constrói o *path* completo do ficheiro.

Depois de iterar sobre todos os ficheiros, a diretoria é fechada. Se um ficheiro com o **ID** especificado for encontrado, a função constrói o *path* completo do ficheiro e envia-o para o cliente. Caso contrário, envia uma mensagem indicando que o ficheiro com o **ID** especificado não foi encontrado. Esta função é crucial para permitir que o cliente aceda ao conteúdo de uma mensagem específica identificada pelo seu **ID**.

Ao receber o *path* do ficheiro, o cliente tentará abri-lo, podendo ter sucesso ou não, dependendo das permissões atribuídas à mensagem.

3.1.4. Responder ao Remetente de uma Mensagem Previamente Recebida

A função `handle_answer_command` (Figura 9) é responsável por criar e enviar uma resposta a uma mensagem específica. Começa por abrir a diretoria onde as mensagens estão armazenadas e itera sobre cada ficheiro procurando pela mensagem original com o **ID** correspondente ao especificado na mensagem recebida. Após encontrar a mensagem original, a função cria um novo ficheiro para a resposta, escreve o conteúdo da resposta nele e ajusta as permissões para que apenas o remetente da resposta possa lê-la. Finalmente, uma mensagem de sucesso é enviada ao cliente indicando que a resposta foi enviada com sucesso. Esta função é essencial para permitir que os utilizadores respondam a mensagens específicas, mantendo a privacidade das respostas.

3.1.5. Apagar uma Mensagem Recebida

A função `handle_delete_command` (Figura 10) é responsável por excluir uma mensagem específica do sistema. Começa por construir o *path* completo do ficheiro da mensagem que se deseja excluir, utilizando a diretoria onde as mensagens são armazenadas, o nome do grupo, o nome do destinatário e o **ID** da mensagem. Seguidamente, verifica se o ficheiro existe e se o utilizador possui as permissões adequadas para excluí-lo (através da função `access()`). Se o ficheiro não existir ou se o utilizador não tiver permissão para o remover, é enviada uma mensagem de erro ao cliente. Se o ficheiro existir e o utilizador tiver permissão para o excluir, a função exclui. Finalmente, uma mensagem de sucesso é enviada ao cliente indicando que a mensagem foi excluída com sucesso. Esta função é fundamental para permitir que os utilizadores tenham controlo sobre as suas mensagens, permitindo que seja possível excluir mensagens específicas quando necessário.

3.1.6. Criar um Grupo de Utilizadores do Serviço

A função `handle_group_create` (Figura 2) é responsável por criar um novo grupo no sistema. Primeiro, constrói um comando *sudo* para adicionar o grupo utilizando `sprintf`. Seguidamente, executa o comando com `system()` e verifica o *status* de retorno. Se o grupo for criado com sucesso, escreve uma mensagem de sucesso no *file descriptor* do *daemon*; caso contrário, escreve uma mensagem de falha.

O criador do grupo é registado num ficheiro de texto (*groups*), permitindo que a sua identidade seja confirmada posteriormente, facilitando a gestão do grupo por parte do seu criador. Esta decisão foi tomada, porque o sistema operativo não guarda a informação relativa ao criador do grupo.

3.1.7. Remover um Grupo de Utilizadores do Serviço

A função `handle_group_delete` (Figura 3) lida com a remoção de um grupo do sistema. Primeiro, obtém o proprietário do grupo com `get_group_owner()`. Se o grupo não existir, escreve uma mensagem de erro informando o utilizador. De seguida, verifica se o remetente da mensagem é o proprietário do grupo (comparando com o resultado obtido na função anterior). Se não for, escreve uma mensagem de erro. Se o remetente for o proprietário, constrói um comando *sudo* para excluir o grupo, executa-o e verifica o *status* de retorno. Se a exclusão for bem-sucedida, remove o grupo do ficheiro de grupos e escreve uma mensagem de sucesso; caso contrário, escreve uma mensagem de falha.

3.1.8. Listar os Utilizadores Membros de um Grupo

A função `handle_group_list` (Figura 11) executa uma listagem dos utilizadores pertencentes a um grupo específico. Constrói um comando para listar os utilizadores do grupo utilizando o comando *getent group* e redireciona o *output* para um ficheiro chamado "temp". Após a execução do comando, verifica-se se ocorreu algum erro. Se sim, uma mensagem de falha é enviada ao cliente. Em caso de sucesso, o ficheiro temporário é aberto e lido linha por linha, com cada linha representando um utilizador do grupo. Os nomes dos utilizadores são adicionados a uma *string* de resposta. Após a leitura completa do ficheiro, é fechado e removido. A *string* de resposta, contendo a lista de utilizadores do grupo, é enviada ao *daemon* para ser transmitida ao cliente. Esta função é útil para permitir que os utilizadores obtenham uma lista dos membros de um grupo específico, facilitando a comunicação e a colaboração entre os membros envolvidos.

3.1.9. Adicionar um Utilizador a um Grupo

A função `handle_group_add` (Figura 4) é responsável por adicionar um utilizador a um grupo existente. Começa por obter o proprietário do grupo utilizando `get_group_owner()`. Se o grupo não existir, uma mensagem de erro é enviada para o *file descriptor* do *daemon* indicando o mesmo. Seguidamente, verifica se o remetente da mensagem é o proprietário do grupo. Se não for, uma mensagem de erro é enviada.

Se o remetente for o proprietário do grupo, a função constrói um comando *sudo* para adicionar o utilizador ao grupo utilizando `sprintf`. Este comando é então executado utilizando `system()`, e o seu *status* de retorno é verificado. Se o utilizador for adicionado com sucesso ao grupo, uma mensagem de sucesso é enviada para o *file descriptor* do *daemon*; caso contrário, uma mensagem de falha é enviada.

Nota: O sistema operativo exige que seja feito *logout* e *login*, para que as alterações sejam feitas.

3.1.10. Remover um Utilizador de um Grupo

A função `handle_group_remove` (Figura 5) é responsável por remover um utilizador de um grupo existente. Primeiro, obtém o proprietário do grupo utilizando `get_group_owner()`. Se o grupo não existir, uma mensagem de erro é enviada para o *file descriptor* do *daemon*.

Seguidamente, verifica se o remetente da mensagem é o proprietário do grupo. Se não for, uma mensagem de erro é enviada.

Se o remetente for o proprietário do grupo, a função constrói um comando *sudo* para remover o utilizador do grupo utilizando `sprintf`. Este comando é então executado utilizando `system()`, e o seu *status* de retorno é verificado. Se o utilizador for removido com sucesso do grupo, uma

mensagem de sucesso é enviada para o *file descriptor* do *daemon*; caso contrário, uma mensagem de falha é enviada.

3.2. Segurança do Serviço

Nesta secção, iremos detalhar as decisões tomadas no domínio da segurança para garantir a integridade, confidencialidade e disponibilidade do serviço. Isto inclui especificações sobre os proprietários e permissões definidas em objetos do sistema de ficheiros, bem como nos processos necessários à execução do serviço.

3.2.1. Proprietários e Permissões dos Ficheiros do Sistema

Os ficheiros do sistema são configurados com permissões de acesso adequadas para garantir a segurança das informações. Eis os detalhes das permissões definidas para os principais objetos do *file system*:

- **Mensagens Armazenadas:** Os ficheiros que armazenam mensagens são configurados com permissões restritas para garantir que apenas o *daemon* tenha capacidade de leitura e de escrita, medida essencial para proteger a confidencialidade das mensagens.

  Os clientes que recebem as mensagens têm permissões de acesso de **0400** e **0440**, o que significa que têm permissão apenas de leitura para os ficheiros de mensagem. No contexto de permissões *Unix*, o número 4 representa permissão de leitura, enquanto que o 0 representa a ausência de permissão para escrever ou executar.

- **Mensagens Lidas e Grupos:** Os ficheiros *read_messages* e *groups* possuem identificadores das mensagens que foram lidas para que não sejam listadas no comando *list*. Todos os utilizadores têm acesso de leitura e nenhum de escrita sobre estes ficheiros, o que não compromete a privacidade das mensagens, dado que o corpo da mensagem, assim como informações possivelmente sensíveis não são apresentadas.

  O ficheiro *groups* também indica o criador do grupo, permitindo assim que se saiba quem terá as permissões de modificar o respetivo grupo.

3.2.2. Proprietários e Permissões dos Processos

Os processos necessários para a execução do serviço também são configurados com permissões adequadas para garantir a segurança do sistema. Aqui estão os detalhes das permissões dos processos:

Daemon: O *daemon* responsável pela gestão das mensagens é executado com privilégios de *root*, medida necessária para que possa aceder e manipular os ficheiros de mensagens no *file system*.

Cliente: O cliente, utilizado pelos utilizadores para interagir com o serviço, é executado com permissões normais de utilizador, limitando as suas capacidades de manipulação do sistema, garantindo que os utilizadores não possam realizar operações que possam comprometer a segurança do sistema.

3.3. Fundamentação da Arquitetura e Desenvolvimento do Sistema de Comunicação

Os aspetos funcionais mais relevantes do sistema incluem a capacidade de enviar, receber e gerir mensagens de forma eficiente e segura, bem como a criação e gestão de grupos de utilizadores. Estas funcionalidades são essenciais para proporcionar uma experiência de comunicação eficaz aos utilizadores. Já os aspetos de segurança mais relevantes incluem o controlo de acesso às mensagens e a autenticação dos utilizadores. Estas medidas são fundamentais para garantir a integridade e confidencialidade das informações trocadas no sistema.

O sistema foi projetado com base em princípios de modularidade e encapsulamento, criando programas secundários para a gestão de mensagens e de grupos, o que facilita a manutenção e evolução do código-fonte. Cada componente de *software* é responsável por uma função específica e bem definida, o que simplifica o desenvolvimento, teste e *debug* do sistema. Além disso, a utilização de interfaces bem definidas entre as componentes permite a substituição ou atualização de partes do sistema sem afetar o código restante.

Durante o desenvolvimento do sistema, foram utilizadas ferramentas para identificar e corrigir problemas no código-fonte, incluindo ferramentas como *debuggers*. Estas ferramentas foram fundamentais para garantir a qualidade e segurança do *software*, identificando potenciais vulnerabilidades e erros de programação.

4. Conclusão

O sistema de mensagens interativo em **C** apresentado neste relatório foi projetado com o objetivo de fornecer uma plataforma de comunicação eficiente e segura para os utilizadores, permitindo o envio, receção e gestão de mensagens individuais e de grupos. As principais decisões arquiteturais incluíram a utilização de *named pipes* para comunicação bidirecional entre o *daemon* e os clientes, garantindo baixa latência e alto desempenho na entrega das mensagens, além da implementação de permissões de acesso e controlo de acesso para garantir a segurança das informações trocadas.

No entanto, algumas limitações e pontos fracos foram identificados durante a análise do sistema, incluindo:

1. **Controlo de acesso limitado:** Embora o sistema utilize permissões de acesso ao *file system* para restringir o acesso às mensagens apenas aos utilizadores autorizados, o controlo poderá ser aprimorado com a implementação de *pipes* com nome individuais para cada utilizador, garantindo permissões específicas para cada *pipe* e reforçando o controlo de acesso.
2. **Falta de criptografia:** O sistema não inclui criptografia nas mensagens gravadas e nas informações transmitidas pelos *pipes*, o que pode comprometer a confidencialidade das comunicações. A implementação de criptografia garantiria que as comunicações fossem protegidas contra acessos não autorizados.
3. **Ausência de comandos de ativação e desativação de utilizadores:** A inclusão de comandos que permitam a ativação e desativação de utilizadores no serviço facilitaria a gestão dos utilizadores e garantiria uma interação mais segura e controlada com o sistema.
4. **Funcionalidade de respostas com referência à mensagem anterior:** A falta de uma funcionalidade que permita aos utilizadores responderem com referência ao conteúdo da mensagem anterior pode dificultar o acompanhamento do contexto das discussões, especialmente em conversas longas ou em grupos. A implementação desta funcionalidade melhoraria a clareza e organização das conversas.
5. **Falta de Restrição de Acesso no *Daemon*:** Até o momento, o *daemon* não foi projetado para restringir as suas capacidades, o que pode resultar numa superfície expandida para potenciais vulnerabilidades de segurança. Embora esta abordagem inicial permita uma maior flexibilidade e adaptabilidade do sistema, também aumenta o risco de exploração por parte de invasores mal-intencionados. A ausência de restrições pode tornar o sistema mais suscetível a ataques e comprometer a segurança das informações trocadas. Portanto, é essencial considerar a implementação de medidas de segurança adicionais para mitigar este risco e garantir a integridade e confidencialidade das comunicações no sistema.

Para superar estas limitações e melhorar o sistema, são propostas as seguintes melhorias:

1. Implementação de *pipes* com nome individuais para cada utilizador.
2. Adição de criptografia nas mensagens e informações transmitidas.
3. Inclusão de comandos de ativação e desativação de utilizadores.

4. Implementação da funcionalidade de respostas com referência à mensagem anterior.
5. Implementação de Restrições de Acesso no *Daemon*: Para abordar a falta de restrição de acesso no *daemon* e fortalecer a segurança do sistema, é recomendável implementar medidas para controlar e limitar as capacidades do *daemon*. Isso pode ser alcançado por através da aplicação de políticas de segurança que definam quais operações o *daemon* pode executar a quais recursos ele pode aceder.

Estas melhorias não apenas abordariam os pontos fracos identificados, mas também elevariam a segurança, eficiência e usabilidade do sistema, proporcionando uma experiência mais positiva aos utilizadores.

5. Anexos

5.1. Daemon.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <dirent.h>
#include <pwd.h>
#include <grp.h>
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <string.h>
#include "../include/message.h"
#include "../include/daemon.h"

int counter = 1;

int get_current_counter()
{
    DIR *dir;
    struct dirent *entry;
    int current_max_id = 0;

    dir = opendir(MESSAGE_FOLDER);
    if (dir == NULL)
    {
        perror("Error opening directory");
        return -1; // Return -1 to indicate error
    }

    while ((entry = readdir(dir)) != NULL)
    {
        // Check if the file name matches the pattern
        if (strstr(entry->d_name, "_") != NULL)
        {
           char *token = strtok(entry->d_name, "_"); // Tokenize by underscore
             token = strtok(NULL, "_");                 // Move to the second
token (message receiver)
             token = strtok(NULL, "_");                 // Move to the third
```

```
token (message ID)
             int id = atoi(token);                      // Convert token to
integer

            // Update current_max_id if necessary
            if (id > current_max_id)
            {
                current_max_id = id;
            }
        }
    }

    closedir(dir);
    return current_max_id;
}

int main()
{
    printf("Make sure daemon is ran as root\n");
    int server_fd_read, server_fd_write;
    char buffer[BUFFER_SIZE];
    counter = get_current_counter();
    memset(buffer, 0, BUFFER_SIZE);

    mkfifo(SERVER_FIFO_READ, 0666);
    mkfifo(SERVER_FIFO_WRITE, 0666);
    // set permissions for all users to be able to read and write
    chmod(SERVER_FIFO_READ, 0666);
    chmod(SERVER_FIFO_WRITE, 0666);

    server_fd_read = open(SERVER_FIFO_READ, O_RDONLY);
    server_fd_write = open(SERVER_FIFO_WRITE, O_WRONLY);
    Message *message = malloc(sizeof(Message));

    while (1)
    {
        if (read(server_fd_read, buffer, BUFFER_SIZE) > 0)
        {
            printf("Buffer: %s\n", buffer);
          printf("Message struct: %d %s %s %s %s %s\n", message->id, message-
>type, message->sender, message->group, message->receiver, message->content);
             sscanf(buffer, "%d %s %s %s %s %[^\n]", &message->id, message-
>type, message->sender, message->group, message->receiver, message->content);

            if (strlen(message->content) > 512)
            {
                printf("Content is too big\n");
                continue;
            }

            if (strcmp(message->type, "write") == 0)
            {
                printf("Content: %s\n", message->content);
             sscanf(buffer, "write %s %s %[^\n]", message->receiver, message-
>sender, message->content);
```

```
            handle_write_command(*message, server_fd_write, counter);
            counter++;
            memset(buffer, 0, BUFFER_SIZE);
        }
        else if (strcmp(message->type, "read") == 0)
        {
            handle_read_command(*message, server_fd_write);

            memset(buffer, 0, BUFFER_SIZE);
        }
        else if (strcmp(message->type, "list") == 0)
        {
            handle_list_command(*message, server_fd_write);

            memset(buffer, 0, BUFFER_SIZE);
        }
        else if (strcmp(message->type, "list_a") == 0)
        {
            handle_listall_command(*message, server_fd_write);

            memset(buffer, 0, BUFFER_SIZE);
        }
        else if (strcmp(message->type, "answer") == 0)
        {
            handle_answer_command(*message, server_fd_write, counter);
            counter++;

            memset(buffer, 0, BUFFER_SIZE);
        }
        else if (strcmp(message->type, "delete") == 0)
        {
            handle_delete_command(*message, server_fd_write);

            memset(buffer, 0, BUFFER_SIZE);
        }
        else if (strcmp(message->type, "group_create") == 0)
        {
            handle_group_create(*message, server_fd_write);

            memset(buffer, 0, BUFFER_SIZE);
        }
        else if (strcmp(message->type, "group_add") == 0)
        {
            handle_group_add(*message, server_fd_write);

            memset(buffer, 0, BUFFER_SIZE);
        }
        else if (strcmp(message->type, "group_remove") == 0)
        {
            handle_group_remove(*message, server_fd_write);

            memset(buffer, 0, BUFFER_SIZE);
        }
        else if (strcmp(message->type, "group_delete") == 0)
```

```
            {
                handle_group_delete(*message, server_fd_write);

                memset(buffer, 0, BUFFER_SIZE);
            }
            else if (strcmp(message->type, "group_list") == 0)
            {
                handle_group_list(*message, server_fd_write);

                memset(buffer, 0, BUFFER_SIZE);
            }
            else
            {
                printf("Invalid command\n");
                write(server_fd_write, "Invalid command\n", 16);
                memset(buffer, 0, BUFFER_SIZE);
            }
            fflush(stdout);
        }
    }

    free(message);
    close(server_fd_read);
    close(server_fd_write);
    unlink(SERVER_FIFO_READ);
    unlink(SERVER_FIFO_WRITE);

    return 0;
}
```

5.2. `Client.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <pwd.h>
#include "../include/message.h"
#include "../include/client.h"

#define SERVER_FIFO_READ "/tmp/server_fifo_read"
#define SERVER_FIFO_WRITE "/tmp/server_fifo_write"

void print_commands()
{
    printf("\nAvailable commands:\n\n");

    // Message commands
    printf("  Messages:\n");
```

```
    printf("    list:                                List non-read messages\n");
    printf("    list -a:                               List all messages\n");
   printf("   write group <group_name> <message>: Send message to a group\n");
    printf("    write user <username> <message>:    Send message to a user\n");
    printf("    read <id>:                             Read messages\n");
    printf("    answer <id> <message>:                 Answer a message\n");
    printf("    delete <id>:                           Delete a message\n");

    // Group commands
    printf("  Group management:\n");
    printf("    group create <group_name>:           Create a group\n");
    printf("    group add <group_name> <client>:    Add user to group\n");
   printf("   group remove <group_name> <client>: Remove user from group\n");
    printf("    group delete <group_name>:           Delete a group\n");
     printf("     group list <group_name>:                List all users in a
group\n");

    // Other commands
    printf("  Other:\n");
    printf("    exit:                                  Exit the program\n");
}

int main()
{
    int server_fd_read, server_fd_write;
    char buffer[BUFFER_SIZE];
    struct passwd *pwd = getpwuid(getuid());
    Message message;

    // Print client's name
    printf("Client name: %s\n", pwd->pw_name);

    // Open server FIFO for writing and reading
    server_fd_write = open(SERVER_FIFO_READ, O_WRONLY);
    server_fd_read = open(SERVER_FIFO_WRITE, O_RDONLY);

    // Command line interface
    while (1)
    {
        // Print available commands
        print_commands();
        printf("\nEnter command: ");
        fgets(buffer, BUFFER_SIZE, stdin);

        if (strncmp(buffer, "write group", 11) == 0)
        {
            strcpy(message.type, "write");
            strcpy(message.sender, pwd->pw_name);
            strcpy(message.receiver, "none");
                   sscanf(buffer,  "write  group  %s  %[^\n]",  message.group,
message.content);
        }
        else if (strncmp(buffer, "write user", 10) == 0)
        {
            strcpy(message.type, "write");
```

```
            strcpy(message.sender, pwd->pw_name);
            strcpy(message.group, "none");
                  sscanf(buffer, "write user %s %[^\n]", message.receiver,
message.content);
        }
        else if (strncmp(buffer, "read", 4) == 0)
        {
            strcpy(message.type, "read");
            strcpy(message.group, "root");
            strcpy(message.receiver, pwd->pw_name);
            strcpy(message.content, "read");
            // the id of the message is the number after the argument read
            message.id = atoi(buffer + 5);
            strcpy(message.sender, pwd->pw_name);
        }
        else if (strncmp(buffer, "answer", 6) == 0)
        {
            strcpy(message.type, "answer");
            strcpy(message.group, "none");
            strcpy(message.receiver, pwd->pw_name);
            sscanf(buffer, "answer %d %[^\n]", &message.id, message.content);
            strcpy(message.sender, pwd->pw_name);
        }
        else if (strncmp(buffer, "delete", 6) == 0)
        {
            strcpy(message.type, "delete");
            strcpy(message.group, "none");
            strcpy(message.receiver, pwd->pw_name);
            message.id = atoi(buffer + 7);
            strcpy(message.sender, pwd->pw_name);
        }
        else if (strncmp(buffer, "list -a", 7) == 0)
        {
            strcpy(message.type, "list_a");
            strcpy(message.group, "none");
            strcpy(message.receiver, pwd->pw_name);
            strcpy(message.content, "list");
            message.id = 0;
            strcpy(message.sender, pwd->pw_name);
        }
        else if (strncmp(buffer, "list", 4) == 0)
        {
            strcpy(message.type, "list");
            strcpy(message.group, "none");
            strcpy(message.receiver, pwd->pw_name);
            strcpy(message.content, "list");
            message.id = 0;
            strcpy(message.sender, pwd->pw_name);
        }
        else if (strncmp(buffer, "group create", 12) == 0)
        {
            strcpy(message.type, "group_create");
            strcpy(message.group, strtok(buffer + 12, " "));
            strcpy(message.receiver, pwd->pw_name);
            strcpy(message.content, "none");
```

```
        message.id = 0;
        strcpy(message.sender, pwd->pw_name);
    }
    else if (strncmp(buffer, "group add", 9) == 0)
    {
        strcpy(message.type, "group_add");
        strcpy(message.group, strtok(buffer + 9, " "));
        strcpy(message.receiver, strtok(NULL, " "));
        message.id = 0;
        strcpy(message.sender, pwd->pw_name);
    }
    else if (strncmp(buffer, "group remove", 12) == 0)
    {
        strcpy(message.type, "group_remove");
        strcpy(message.group, strtok(buffer + 12, " "));
        strcpy(message.receiver, strtok(NULL, " "));
        strcpy(message.content, "none");
        message.id = 0;
        strcpy(message.sender, pwd->pw_name);
    }
    else if (strncmp(buffer, "group delete", 12) == 0)
    {
        strcpy(message.type, "group_delete");
        strcpy(message.group, strtok(buffer + 12, " "));
        strcpy(message.receiver, pwd->pw_name);
        strcpy(message.content, "none");
        message.id = 0;
        strcpy(message.sender, pwd->pw_name);
    }
    else if (strncmp(buffer, "group list", 10) == 0)
    {
        strcpy(message.type, "group_list");
        strcpy(message.group, strtok(buffer + 10, " "));
        strcpy(message.receiver, pwd->pw_name);
        strcpy(message.content, "none");
        message.id = 0;
        strcpy(message.sender, pwd->pw_name);
    }
    else if (strncmp(buffer, "exit", 4) == 0)
    {
        return;
        break;
    }
    else
    {
        printf("Invalid command\n");
        continue;
    }

    // Write the message to the server
    write_message(message, server_fd_write);

    // Clear the buffer
    memset(buffer, 0, BUFFER_SIZE);
```

```
        // Read the response from the server
        read(server_fd_read, buffer, BUFFER_SIZE);
        if (strncmp(buffer, "messages/", 9) == 0)
        {
            printf("Buffer: %s\n", buffer);
            char *file_path = buffer;
            read_message(file_path);
        }
        else
        {
            printf("Daemon response:\n%s\n", buffer);
        }
    }

    // Close server FIFO
    close(server_fd_read);
    close(server_fd_write);

    return 0;
}
```

5.3. Group.c

```
#include "../include/group.h"
#include <stdio.h>
#include <unistd.h>
#include <string.h>

char *get_group_owner(char *group_name)
{
  char file_path[512];
  sprintf(file_path, "files/groups", MESSAGE_FOLDER);
  FILE *file = fopen(file_path, "r");
  if (file == NULL)
  {
    perror("Error opening file");
    return NULL;
  }

  char *line = NULL;
  size_t len = 0;
  ssize_t read;
  char *owner = NULL;

  while ((read = getline(&line, &len, file)) != -1)
  {
    char *group = strtok(line, "$");
    char *user = strtok(NULL, "$");

    if (strcmp(group, group_name) == 0)
    {
```

```
      owner = user;
      break;
    }
  }

  fclose(file);
  return owner;
}

void write_groups_file(char *group_name, char *user)
{
  // write to file group$user and if the file already exists, append to it
  char file_path[512];
  // file path is "../files/groups"
  sprintf(file_path, "files/groups", MESSAGE_FOLDER);
  FILE *file = fopen(file_path, "a");
  if (file == NULL)
  {
    perror("Error opening file");
    return;
  }
  fprintf(file, "%s$%s\n", group_name, user);
  fclose(file);
}

void delete_group_groups_file(char *group_name)
{
  char file_path[512];
  sprintf(file_path, "files/groups", MESSAGE_FOLDER);
  FILE *file = fopen(file_path, "r");
  if (file == NULL)
  {
    perror("Error opening file");
    return;
  }

  char *line = NULL;
  size_t len = 0;
  ssize_t read;
  char *owner = NULL;

  FILE *temp = fopen("temp", "w");
  while ((read = getline(&line, &len, file)) != -1)
  {
    char *group = strtok(line, "$");
    char *user = strtok(NULL, "$");

    if (strcmp(group, group_name) != 0)
    {
      fprintf(temp, "%s$%s\n", group, user);
    }
  }

  fclose(file);
  fclose(temp);
```

```
  remove(file_path);
  rename("temp", file_path);
}

void handle_group_create(Message message, int server_fd_write)
{
  // Construct the command to create the group using system()
  char command[100];
  sprintf(command, "sudo groupadd %s", message.group);

  // Execute the command
  int status = system(command);
  if (status == 0)
  {
    write_groups_file(message.group, message.sender);
    char success_message[15];
    strcpy(success_message, "Group created.\n");
    write(server_fd_write, success_message, strlen(success_message));
  }
  else
  {
    char failure_message[15];
    strcpy(failure_message, "Unable to create group.\n");
    write(server_fd_write, failure_message, strlen(failure_message));
  }
}

void handle_group_add(Message message, int server_fd_write)
{
  char *owner = get_group_owner(message.group);
  if (owner == NULL)
  {
    char error_message[50];
    sprintf(error_message, "Group %s does not exist\n", message.group);
    write(server_fd_write, error_message, strlen(error_message));
    return;
  }
  owner[strcspn(owner, "\n")] = 0;
  owner[strcspn(owner, " ")] = 0;
  message.receiver[strcspn(message.receiver, "\n")] = 0;
  message.receiver[strcspn(message.receiver, " ")] = 0;

  if (strcmp(owner, message.sender) != 0)
  {
    char error_message[50];
      sprintf(error_message, "You are not the owner of the group %s\n",
message.group);
    write(server_fd_write, error_message, strlen(error_message));
    return;
  }
  // execute command to add user to group
  char command[100];
     sprintf(command, "sudo usermod -a -G %s %s", message.group,
message.receiver);
  int status = system(command);
```

```
  if (status == 0)
  {
    char success_message[15];
    strcpy(success_message, "User added.\n");
    write(server_fd_write, success_message, strlen(success_message));
  }
  else
  {
    char failure_message[15];
    strcpy(failure_message, "Unable to add user.\n");
    write(server_fd_write, failure_message, strlen(failure_message));
  }
}

void handle_group_remove(Message message, int server_fd_write)
{
  char *owner = get_group_owner(message.group);
  if (owner == NULL)
  {
    char error_message[50];
    sprintf(error_message, "Group %s does not exist\n", message.group);
    write(server_fd_write, error_message, strlen(error_message));
    return;
  }
  owner[strcspn(owner, "\n")] = 0;
  owner[strcspn(owner, " ")] = 0;
  message.sender[strcspn(message.sender, "\n")] = 0;
  message.sender[strcspn(message.sender, " ")] = 0;

  if (strcmp(owner, message.sender) != 0)
  {
    char error_message[50];
      sprintf(error_message, "You are not the owner of the group %s\n",
message.group);
    write(server_fd_write, error_message, strlen(error_message));
    return;
  }
  // execute command to remove user from group
  char command[100];
  sprintf(command, "sudo gpasswd -d %s %s", message.receiver, message.group);
  int status = system(command);
  if (status != 0)
  {
    char failure_message[15];
    strcpy(failure_message, "Unable to remove user.\n");
    write(server_fd_write, failure_message, strlen(failure_message));
    return;
  }
  char success_message[15];
  strcpy(success_message, "User removed.\n");
  write(server_fd_write, success_message, strlen(success_message));
}

void handle_group_delete(Message message, int server_fd_write)
{
```

```
  char *owner = get_group_owner(message.group);
  if (owner == NULL)
  {
    char error_message[50];
    sprintf(error_message, "Group %s does not exist\n", message.group);
    write(server_fd_write, error_message, strlen(error_message));
    return;
  }
  owner[strcspn(owner, "\n")] = 0;
  owner[strcspn(owner, " ")] = 0;
  message.sender[strcspn(message.sender, "\n")] = 0;
  message.sender[strcspn(message.sender, " ")] = 0;
  if (strcmp(owner, message.sender) != 0)
  {
    char error_message[50];
      sprintf(error_message, "You are not the owner of the group %s\n",
message.group);
    write(server_fd_write, error_message, strlen(error_message));
    return;
  }

  // Construct the command to create the group using system()
  char command[100];
  sprintf(command, "sudo groupdel %s", message.group);
  int status = system(command);
  if (status == 0)
  {
    delete_group_groups_file(message.group);
    char success_message[15];
    strcpy(success_message, "Group delete.\n");
    write(server_fd_write, success_message, strlen(success_message));
  }
  else
  {
    char failure_message[15];
    strcpy(failure_message, "Unable to delete group.\n");
    write(server_fd_write, failure_message, strlen(failure_message));
  }
}

void handle_group_list(Message message, int server_fd_write)
{
  // execute command to list users in group and save it to a file
  char command[100];
 sprintf(command, "sudo getent group %s | cut -d: -f4 > temp", message.group);
  int status = system(command);
  if (status != 0)
  {
    char failure_message[15];
    strcpy(failure_message, "Unable to list users.\n");
    write(server_fd_write, failure_message, strlen(failure_message));
    return;
  }
  // read the file and send the content to the client
  FILE *file = fopen("temp", "r");
```

```
  if (file == NULL)
  {
    perror("Error opening file");
    write(server_fd_write, "Error listing users\n", 20);
    return;
  }
  char response[BUFFER_SIZE];
  strcpy(response, "List of users:\n");
  char user[50];
  while (fscanf(file, "%s", user) != EOF)
  {
    sprintf(response + strlen(response), "%s\n", user);
  }
  fclose(file);
  remove("temp");
  write(server_fd_write, response, strlen(response));
}
```

5.4. Message.c

```
#include "../include/message.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <grp.h>
#include <pwd.h>
#include <dirent.h>
#include <unistd.h>
#include "../include/daemon.h"
void write_message(Message message, int server_fd_write)
{
  // Serialize the message
  char serialized_message[BUFFER_SIZE];
  sprintf(serialized_message, "%d %s %s %s %s %s", message.id, message.type,
message.sender, message.group, message.receiver, message.content);

  // Write the serialized message to the server
  write(server_fd_write, serialized_message, strlen(serialized_message));
}

void *read_message(char *file_path)
{
  FILE *file = fopen(file_path, "r");
  char *buffer = malloc(BUFFER_SIZE);
  memset(buffer, 0, BUFFER_SIZE);

  if (file == NULL)
  {
    printf("Error opening file, check permissions.\n\n");
    return NULL;
  }
```

```
  fread(buffer, 1, BUFFER_SIZE, file);
  fclose(file);

  // separate message in buffer by $
  char *token = strtok(buffer, "$");
  printf("Sender: %s\n", token);
  token = strtok(NULL, "$");
  printf("Message: %s\n", token);
}

void handle_write_command(Message message, int server_fd_write, int counter)
{
  char file_path[512];
      sprintf(file_path,   "%s/%s_%s_%d",   MESSAGE_FOLDER,   message.group,
message.receiver, counter);

  printf("File path: %s\n", file_path);
  // set permissions for everyone to not be able to read and write
  chmod(file_path, 0000);
  // get receiver uid
  struct passwd *pwd = getpwnam(message.receiver);
  // get receiver gid
  struct group *grp = getgrnam(message.group);

  // check if the user exists
  if (pwd == NULL && grp == NULL)
  {
    char error_message[50];
    sprintf(error_message, "Does not exist\n");
    write(server_fd_write, error_message, strlen(error_message));
    return;
  }
  // if they are null set as no user and no group
  int uid = 0;
  int gid = 0;
  if (pwd == NULL)
  {
    gid = grp->gr_gid;
  }
  if (grp == NULL)
  {
    uid = pwd->pw_uid;
  }

  printf("Receiver uid: %d\n", uid);
  printf("Receiver gid: %d\n", gid);

  // Create the messages directory if it doesn't exist
  mkdir(MESSAGE_FOLDER, 0777);

  FILE *file = fopen(file_path, "w");
  if (file == NULL)
  {
    perror("Error opening file\n");
```

```
    return;
  }

  fprintf(file, "%s$%s\n", message.sender, message.content);

  fclose(file);

  // now give permissions to the user the is the receiver
  chown(file_path, uid, gid);
  // change permissions for owner
  if (pwd != NULL)
  {
    chmod(file_path, 0600);
  }
  // change permissions for group
  if (grp != NULL)
  {
    chmod(file_path, 0660);
  }
  char success_message[15];
  strcpy(success_message, "Message sent.\n");
  write(server_fd_write, success_message, strlen(success_message));
}
void handle_read_command(Message message, int server_fd_write)
{
  DIR *dir;
  struct dirent *entry;
  char response[BUFFER_SIZE];
  char file_path[512];
  int found = 0;

  // Open the directory
  dir = opendir(MESSAGE_FOLDER);
  if (dir == NULL)
  {
    perror("Error opening directory");
    write(server_fd_write, "Error reading messages\n", 23);
    return;
  }

  // Iterate over each entry in the directory
  while ((entry = readdir(dir)) != NULL)
  {
    // Check if the file name matches the pattern
    if (strstr(entry->d_name, "_") != NULL)
    {
      strcpy(response, entry->d_name);
      // Extract the ID from the file name
      char *token = strtok(entry->d_name, "_");
      for (int i = 0; i < 2; i++)
      {
        token = strtok(NULL, "_");
      }
      int id = atoi(token);
```

```
      // Check if the ID matches the specified ID
      if (id == message.id)
      {
        found = 1;
        // Construct full file path
        sprintf(file_path, "%s/%s", MESSAGE_FOLDER, entry->d_name);
        break;
      }
    }
  }

  // Close the directory
  closedir(dir);

  // If file with the specified ID is found, send its path
  if (found)
  {
    // preppend the "messages/" to the file path
    char full_file_path[512];
    strcpy(full_file_path, "messages/");
    strcat(full_file_path, response);
    write(server_fd_write, full_file_path, strlen(full_file_path));

    // Append the file name to the read_messages file
    FILE *read_messages_file = fopen("files/read_messages", "a");
    if (read_messages_file == NULL)
    {
      perror("Error opening read_messages file\n");
      return;
    }
    fprintf(read_messages_file, "%s\n", response);
    fclose(read_messages_file);
  }
  else
  {
    sprintf(response, "File with ID %d not found\n", message.id);
    write(server_fd_write, response, strlen(response));
  }
}

void handle_list_command(Message message, int server_fd_write)
{
  DIR *dir;
  struct dirent *entry;
  char response[BUFFER_SIZE];
  char file_path[512];
  char sender[50];
  char strid[10];
  char *ptr;

  dir = opendir(MESSAGE_FOLDER);
  if (dir == NULL)
  {
    perror("Error opening directory");
    write(server_fd_write, "Error listing messages\n", 23);
```

```
    return;
  }

  strcpy(response, "List of messages:\n");
  while ((entry = readdir(dir)) != NULL)
  {
    char filename[512];
    strcpy(filename, entry->d_name);
    // Check if the file name matches the pattern
    if (strstr(entry->d_name, message.sender) != NULL)
    {
      // Extract sender from the file (file has sender$message)
      sprintf(file_path, "%s/%s", MESSAGE_FOLDER, entry->d_name);
      FILE *file = fopen(file_path, "r");
      if (file == NULL)
      {
        perror("Error opening file");
        write(server_fd_write, "Error listing messages\n", 23);
        return;
      }
      fscanf(file, "%[^$]", sender);
      strtok_r(sender, "_", &ptr); // Remove receivergroup

      // Tokenize file name and set id to the value at position 2
      char *token = strtok(entry->d_name, "_");
      for (int i = 0; i < 2; i++)
      {
        token = strtok(NULL, "_");
      }
      int id = atoi(token);
      // check if entry is inside the read_messages file
      FILE *read_messages_file = fopen("files/read_messages", "r");
      if (read_messages_file == NULL)
      {
        perror("Error opening read_messages file\n");
        return;
      }
      char line[512];
      int found = 0;
      while (fgets(line, sizeof(line), read_messages_file))
      {
        printf("Filename: %s\n", filename);
        printf("Line: %s\n", line);
        if (strstr(line, filename) != NULL)
        {
          found = 1;
          break;
        }
      }
      // Append sender and ID to response string
       if(found != 1) sprintf(response + strlen(response), "Sender: %s, ID:
%d\n", sender, id);
    }
  }
  closedir(dir);
```

```
  // Send the response to the client
  write(server_fd_write, response, strlen(response));
}

void handle_listall_command(Message message, int server_fd_write)
{
  DIR *dir;
  struct dirent *entry;
  char response[BUFFER_SIZE];
  char file_path[512];
  char sender[50];
  char strid[10];
  char *ptr;

  dir = opendir(MESSAGE_FOLDER);
  if (dir == NULL)
  {
    perror("Error opening directory");
    write(server_fd_write, "Error listing messages\n", 23);
    return;
  }

  strcpy(response, "List of messages:\n");
  while ((entry = readdir(dir)) != NULL)
  {
    // Check if the file name matches the pattern
    if (strstr(entry->d_name, message.sender) != NULL)
    {
      // Extract sender from the file (file has sender$message)
      sprintf(file_path, "%s/%s", MESSAGE_FOLDER, entry->d_name);
      FILE *file = fopen(file_path, "r");
      if (file == NULL)
      {
        perror("Error opening file");
        write(server_fd_write, "Error listing messages\n", 23);
        return;
      }
      fscanf(file, "%[^$]", sender);
      strtok_r(sender, "_", &ptr); // Remove receivergroup

      // Tokenize file name and set id to the value at position 2
      char *token = strtok(entry->d_name, "_");
      for (int i = 0; i < 2; i++)
      {
        token = strtok(NULL, "_");
      }
      int id = atoi(token);

      // Append sender and ID to response string
       sprintf(response + strlen(response), "Sender: %s, ID: %d\n", sender,
id);
    }
  }
  closedir(dir);
```

```
  // Send the response to the client
  write(server_fd_write, response, strlen(response));
}

void handle_answer_command(Message message, int server_fd_write, int counter)
{
  // create new message for sender of the one you're answering with content
  // find file with specified id and get the sender
  DIR *dir;
  struct dirent *entry;
  char response[BUFFER_SIZE];
  char file_path[512];
  char sender[50];
  char strid[10];
  char *ptr;

  dir = opendir(MESSAGE_FOLDER);
  if (dir == NULL)
  {
    perror("Error opening directory");
    write(server_fd_write, "Error listing messages\n", 23);
    return;
  }

  while ((entry = readdir(dir)) != NULL)
  {
    // Check if the file name matches the pattern
    if (strstr(entry->d_name, message.sender) != NULL)
    {
      // Extract sender from the file (file has sender$message)
      sprintf(file_path, "%s/%s", MESSAGE_FOLDER, entry->d_name);
      FILE *file = fopen(file_path, "r");
      if (file == NULL)
      {
        perror("Error opening file");
        write(server_fd_write, "Error listing messages\n", 23);
        return;
      }
      fscanf(file, "%[^$]", sender);
      strtok_r(sender, "_", &ptr); // Remove receivergroup

      // Tokenize file name and set id to the value at position 2
      char *token = strtok(entry->d_name, "_");
      for (int i = 0; i < 2; i++)
      {
        token = strtok(NULL, "_");
      }
      int id = atoi(token);

      // Check if the ID matches the specified ID
      if (id == message.id)
      {
        // Construct full file path
        sprintf(file_path, "%s/%s", MESSAGE_FOLDER, entry->d_name);
```

```
        break;
      }
    }
  }

  closedir(dir);

  // create new message file
  char new_file_path[512];
    sprintf(new_file_path,  "%s/%s_%s_%d",  MESSAGE_FOLDER,  "none",  sender,
counter);
  FILE *new_file = fopen(new_file_path, "w");
  if (new_file == NULL)
  {
    perror("Error opening file\n");
    return;
  }
  fprintf(new_file, "%s$%s\n", message.sender, message.content);
  fclose(new_file);
  // set permissions for everyone to not be able to read and write
  chmod(new_file_path, 0000);
  // get receiver uid
  struct passwd *pwd = getpwnam(sender);
  // get receiver gid
  struct group *grp = getgrnam(message.group);

  // check if the user exists
  if (pwd == NULL && grp == NULL)
  {
    char error_message[50];
    sprintf(error_message, "Does not exist\n");
    write(server_fd_write, error_message, strlen(error_message));
    return;
  }
  // if they are null set as no user and no group
  int uid = 0;
  int gid = 0;

  printf("Sender: %s\n", sender);
  uid = pwd->pw_uid;
  printf("Receiver uid: %d\n", uid);

  // now give permissions to the user the is the receiver
  chown(new_file_path, uid, gid);
  chmod(new_file_path, 0600);
  chmod(new_file_path, 0660);

  char success_message[15];
  strcpy(success_message, "Message sent.\n");
  write(server_fd_write, success_message, strlen(success_message));
}

void handle_delete_command(Message message, int server_fd_write)
{
  char file_path[512];
```

```
      sprintf(file_path,   "%s/%s_%s_%d",   MESSAGE_FOLDER,   message.group,
message.receiver, message.id);

  // Check if the file exists
  if (access(file_path, F_OK) == -1)
  {
    char error_message[50];
       sprintf(error_message,  "File  with  ID  %d  not  found  or  permissions
insufficient.\n", message.id);
    write(server_fd_write, error_message, strlen(error_message));
    return;
  }

  // Delete the file
  remove(file_path);

  char success_message[15];
  strcpy(success_message, "Message deleted.\n");
  write(server_fd_write, success_message, strlen(success_message));
}
```

5.5. Imagens

```
1  void handle_group_create(Message message, int server_fd_write)
2  {
3      // Construct the command to create the group using system()
4      char command[100];
5      sprintf(command, "sudo groupadd %s", message.group);
6
7      // Execute the command
8      int status = system(command);
9      if (status == 0)
10     {
11         write_groups_file(message.group, message.sender);
12         char success_message[15];
13         strcpy(success_message, "Group created.\n");
14         write(server_fd_write, success_message, strlen(success_message));
15     }
16     else
17     {
18         char failure_message[15];
19         strcpy(failure_message, "Unable to create group.\n");
20         write(server_fd_write, failure_message, strlen(failure_message));
21     }
22 }
```

Figura 2: Função `handle_group_create()`

```c
void handle_group_delete(Message message, int server_fd_write)
{
    char *owner = get_group_owner(message.group);
    if (owner == NULL)
    {
        char error_message[50];
        sprintf(error_message, "Group %s does not exist\n", message.group);
        write(server_fd_write, error_message, strlen(error_message));
        return;
    }
    owner[strcspn(owner, "\n")] = 0;
    owner[strcspn(owner, " ")] = 0;
    message.sender[strcspn(message.sender, "\n")] = 0;
    message.sender[strcspn(message.sender, " ")] = 0;
    if(strcmp(owner, message.sender) != 0)
    {
        char error_message[50];
        sprintf(error_message, "You are not the owner of the group %s\n", message.group);
        write(server_fd_write, error_message, strlen(error_message));
        return;
    }

    // Construct the command to create the group using system()
    char command[100];
    sprintf(command, "sudo groupdel %s", message.group);
    int status = system(command);
    if (status == 0)
    {
        delete_group_groups_file(message.group);
        char success_message[15];
        strcpy(success_message, "Group delete.\n");
        write(server_fd_write, success_message, strlen(success_message));
    }
    else
    {
        char failure_message[15];
        strcpy(failure_message, "Unable to delete group.\n");
        write(server_fd_write, failure_message, strlen(failure_message));
    }
}

```

Figura 3: Função `handle_group_delete()`

```c
void handle_group_add(Message message, int server_fd_write)
{
    char *owner = get_group_owner(message.group);
    if (owner == NULL)
    {
        char error_message[50];
        sprintf(error_message, "Group %s does not exist\n", message.group);
        write(server_fd_write, error_message, strlen(error_message));
        return;
    }
    owner[strcspn(owner, "\n")] = 0;
    owner[strcspn(owner, " ")] = 0;
    message.receiver[strcspn(message.receiver, "\n")] = 0;
    message.receiver[strcspn(message.receiver, " ")] = 0;

    if (strcmp(owner, message.sender) != 0)
    {
        char error_message[50];
        sprintf(error_message, "You are not the owner of the group %s\n", message.group);
        write(server_fd_write, error_message, strlen(error_message));
        return;
    }
    // execute command to add user to group
    char command[100];
    sprintf(command, "sudo usermod -a -G %s %s", message.group, message.receiver);
    int status = system(command);
    if (status == 0)
    {
        char success_message[15];
        strcpy(success_message, "User added.\n");
        write(server_fd_write, success_message, strlen(success_message));
    }
    else
    {
        char failure_message[15];
        strcpy(failure_message, "Unable to add user.\n");
        write(server_fd_write, failure_message, strlen(failure_message));
    }
}
```

Figura 4: Função `handle_group_add()`

```c
void handle_group_remove(Message message, int server_fd_write)
{
    char *owner = get_group_owner(message.group);
    if (owner == NULL)
    {
        char error_message[50];
        sprintf(error_message, "Group %s does not exist\n", message.group);
        write(server_fd_write, error_message, strlen(error_message));
        return;
    }
    owner[strcspn(owner, "\n")] = 0;
    owner[strcspn(owner, " ")] = 0;
    message.sender[strcspn(message.sender, "\n")] = 0;
    message.sender[strcspn(message.sender, " ")] = 0;

    if (strcmp(owner, message.sender) != 0)
    {
        char error_message[50];
        sprintf(error_message, "You are not the owner of the group %s\n", message.group);
        write(server_fd_write, error_message, strlen(error_message));
        return;
    }
    // execute command to remove user from group
    char command[100];
    sprintf(command, "sudo gpasswd -d %s %s", message.receiver, message.group);
    int status = system(command);
    if (status != 0)
    {
        char failure_message[15];
        strcpy(failure_message, "Unable to remove user.\n");
        write(server_fd_write, failure_message, strlen(failure_message));
        return;
    }
    char success_message[15];
    strcpy(success_message, "User removed.\n");
    write(server_fd_write, success_message, strlen(success_message));
}

```

Figura 5: Função `handle_group_remove()`

```c
void handle_write_command(Message message, int server_fd_write)
{
    char file_path[512];
    sprintf(file_path, "%s/%s_%s_%d", MESSAGE_FOLDER, message.group, message.receiver, counter);

    printf("File path: %s\n", file_path);
    // set permissions for everyone to not be able to read and write
    chmod(file_path, 0000);
    // get receiver uid
    struct passwd *pwd = getpwnam(message.receiver);
    // get receiver gid
    struct group *grp = getgrnam(message.group);

    // check if the user exists
    if (pwd == NULL && grp == NULL)
    {
        char error_message[50];
        sprintf(error_message, "Does not exist\n");
        write(server_fd_write, error_message, strlen(error_message));
        return;
    }
    // if they are null set as no user and no group
    int uid = 0;
    int gid = 0;
    if(pwd == NULL)
    {
        gid = grp->gr_gid;
    }
    if(grp == NULL)
    {
        uid = pwd->pw_uid;
    }

    printf("Receiver uid: %d\n", uid);
    printf("Receiver gid: %d\n", gid);

    // Create the messages directory if it doesn't exist
    mkdir(MESSAGE_FOLDER, 0777);

    FILE *file = fopen(file_path, "w");
    if (file == NULL)
    {
        perror("Error opening file\n");
        return;
    }

    fprintf(file, "%s$%s\n", message.sender, message.content);

    fclose(file);

    // now give permissions to the user the is the receiver
    chown(file_path, uid, gid);
    // change permissions for owner
    if(pwd != NULL)
    {
        chmod(file_path, 0600);
    }
    // change permissions for group
    if(grp != NULL)
    {
        chmod(file_path, 0660);
    }
    char success_message[15];
    strcpy(success_message, "Message sent.\n");
    write(server_fd_write, success_message, strlen(success_message));
    counter++;
}
```

Figura 6: Função `handle_write_command()`

```c
void handle_read_command(Message message, int server_fd_write)
{
    DIR *dir;
    struct dirent *entry;
    char response[BUFFER_SIZE];
    char file_path[512];
    int found = 0;

    // Open the directory
    dir = opendir(MESSAGE_FOLDER);
    if (dir == NULL)
    {
        perror("Error opening directory");
        write(server_fd_write, "Error reading messages\n", 23);
        return;
    }

    // Iterate over each entry in the directory
    while ((entry = readdir(dir)) != NULL)
    {
        // Check if the file name matches the pattern
        if (strstr(entry->d_name, "_") != NULL)
        {
            strcpy(response, entry->d_name);
            // Extract the ID from the file name
            char *token = strtok(entry->d_name, "_");
            for (int i = 0; i < 2; i++)
            {
                token = strtok(NULL, "_");
            }
            int id = atoi(token);

            // Check if the ID matches the specified ID
            if (id == message.id)
            {
                found = 1;
                // Construct full file path
                sprintf(file_path, "%s/%s", MESSAGE_FOLDER, entry->d_name);
                break;
            }
        }
    }

    // Close the directory
    closedir(dir);

    // If file with the specified ID is found, send its path
    if(found)
    {
        // preppend the "messages/" to the file path
        char full_file_path[512];
        strcpy(full_file_path, "messages/");
        strcat(full_file_path, response);
        write(server_fd_write, full_file_path, strlen(full_file_path));
    }
    else
    {
        sprintf(response, "File with ID %d not found\n", message.id);
        write(server_fd_write, response, strlen(response));
    }
}
```

Figura 7: Função `handle_read_command()`

```c
void handle_list_command(Message message, int server_fd_write)
{
    DIR *dir;
    struct dirent *entry;
    char response[BUFFER_SIZE];
    char file_path[512];
    char sender[50];
    char strid[10];
    char *ptr;

    dir = opendir(MESSAGE_FOLDER);
    if (dir == NULL)
    {
        perror("Error opening directory");
        write(server_fd_write, "Error listing messages\n", 23);
        return;
    }

    strcpy(response, "List of messages:\n");
    while ((entry = readdir(dir)) != NULL)
    {
        // Check if the file name matches the pattern
        if (strstr(entry->d_name, message.sender) != NULL)
        {
            // Extract sender from the file (file has sender$message)
            sprintf(file_path, "%s/%s", MESSAGE_FOLDER, entry->d_name);
            FILE *file = fopen(file_path, "r");
            if (file == NULL)
            {
                perror("Error opening file");
                write(server_fd_write, "Error listing messages\n", 23);
                return;
            }
            fscanf(file, "%[^$]", sender);
            strtok_r(sender, "_", &ptr); // Remove receivergroup

            // Tokenize file name and set id to the value at position 2
            char *token = strtok(entry->d_name, "_");
            for (int i = 0; i < 2; i++)
            {
                token = strtok(NULL, "_");
            }
            int id = atoi(token);


            // Append sender and ID to response string
            sprintf(response + strlen(response), "Sender: %s, ID: %d\n", sender, id);
        }
    }
    closedir(dir);

    // Send the response to the client
    write(server_fd_write, response, strlen(response));
}
```

Figura 8: Função `handle_list_command()`

```c
void handle_answer_command(Message message, int server_fd_write, int counter)
{
  // create new message for sender of the one you're answering with content
  // find file with specified id and get the sender
  DIR *dir;
  struct dirent *entry;
  char response[BUFFER_SIZE];
  char file_path[512];
  char sender[50];
  char strid[10];
  char *ptr;

  dir = opendir(MESSAGE_FOLDER);
  if (dir == NULL)
  {
    perror("Error opening directory");
    write(server_fd_write, "Error listing messages\n", 23);
    return;
  }

  while ((entry = readdir(dir)) != NULL)
  {
    // Check if the file name matches the pattern
    if (strstr(entry->d_name, message.sender) != NULL)
    {
      // Extract sender from the file (file has sender$message)
      sprintf(file_path, "%s/%s", MESSAGE_FOLDER, entry->d_name);
      FILE *file = fopen(file_path, "r");
      if (file == NULL)
      {
        perror("Error opening file");
        write(server_fd_write, "Error listing messages\n", 23);
        return;
      }
      fscanf(file, "%[^$]", sender);
      strtok_r(sender, "_", &ptr); // Remove receivergroup

      // Tokenize file name and set id to the value at position 2
      char *token = strtok(entry->d_name, "_");
      for (int i = 0; i < 2; i++)
      {
        token = strtok(NULL, "_");
      }
      int id = atoi(token);

      // Check if the ID matches the specified ID
      if (id == message.id)
      {
        // Construct full file path
        sprintf(file_path, "%s/%s", MESSAGE_FOLDER, entry->d_name);
        break;
      }
    }
  }

  closedir(dir);

  // create new message file
  char new_file_path[512];
  sprintf(new_file_path, "%s/%s_%s_%d", MESSAGE_FOLDER, "none", sender, counter);
  FILE *new_file = fopen(new_file_path, "w");
  if (new_file == NULL)
  {
    perror("Error opening file\n");
    return;
  }
  fprintf(new_file, "%s$%s\n", message.sender, message.content);
  fclose(new_file);
  // set permissions for everyone to not be able to read and write
  chmod(new_file_path, 0000);
  // get receiver uid
  struct passwd *pwd = getpwnam(sender);
  // get receiver gid
  struct group *grp = getgrnam(message.group);

  // check if the user exists
  if (pwd == NULL && grp == NULL)
  {
    char error_message[50];
    sprintf(error_message, "Does not exist\n");
    write(server_fd_write, error_message, strlen(error_message));
    return;
  }
  // if they are null set as no user and no group
  int uid = 0;
  int gid = 0;

  printf("Sender: %s\n", sender);
  uid = pwd->pw_uid;
  printf("Receiver uid: %d\n", uid);

  // now give permissions to the user the is the receiver
  chown(new_file_path, uid, gid);
  chmod(new_file_path, 0600);
  chmod(new_file_path, 0660);

  char success_message[15];
  strcpy(success_message, "Message sent.\n");
  write(server_fd_write, success_message, strlen(success_message));
}
```

Figura 9: Função `handle_answer_command()`

```c
void handle_delete_command(Message message, int server_fd_write)
{
  char file_path[512];
  sprintf(file_path, "%s/%s_%s_%d", MESSAGE_FOLDER, message.group, message.receiver, message.id);

  // Check if the file exists
  if (access(file_path, F_OK) == -1)
  {
    char error_message[50];
    sprintf(error_message, "File with ID %d not found or permissions insufficient.\n", message.id);
    write(server_fd_write, error_message, strlen(error_message));
    return;
  }

  // Delete the file
  remove(file_path);

  char success_message[15];
  strcpy(success_message, "Message deleted.\n");
  write(server_fd_write, success_message, strlen(success_message));
}
```

Figura 10: Função `handle_delete_command()`

```c
void handle_group_list(Message message, int server_fd_write)
{
  // execute command to list users in group and save it to a file
  char command[100];
  sprintf(command, "sudo getent group %s | cut -d: -f4 > temp", message.group);
  int status = system(command);
  if (status != 0)
  {
    char failure_message[15];
    strcpy(failure_message, "Unable to list users.\n");
    write(server_fd_write, failure_message, strlen(failure_message));
    return;
  }
  // read the file and send the content to the client
  FILE *file = fopen("temp", "r");
  if (file == NULL)
  {
    perror("Error opening file");
    write(server_fd_write, "Error listing users\n", 20);
    return;
  }
  char response[BUFFER_SIZE];
  strcpy(response, "List of users:\n");
  char user[50];
  while (fscanf(file, "%s", user) != EOF)
  {
    sprintf(response + strlen(response), "%s\n", user);
  }
  fclose(file);
  remove("temp");
  write(server_fd_write, response, strlen(response));
}
```

Figura 11: Função `handle_group_list()`