Relatório Trabalho 3
Comunicação Inter-processos usando Sockets Unix

Arthur Jacobs – 205980
João Victor Gomes Cachola - 218762

1 DESCRIÇÃO DOS AMBIENTES DE TESTE

- Ambiente de teste 1
  - Sistema Operacional e distribuição

```
joao@joao-Inspiron-7520:~$ uname -or
3.16.0-34-generic GNU/Linux
joao@joao-Inspiron-7520:~$ lsb_release -irc
Distributor ID: Ubuntu
Release:        14.10
Codename:       utopic
joao@joao-Inspiron-7520:~$
```

  - Compilador utilizado

```
ii  gcc                                                  4:4.9.1-4ubuntu2
           amd64        GNU C compiler
ii  gcc-4.9                                              4.9.1-16ubuntu6
           amd64        GNU C compiler
```

  - Configurações da máquina

```
joao@joao-Inspiron-7520:~$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    2
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 58
Model name:            Intel(R) Core(TM) i7-3612QM CPU @ 2.10GHz
Stepping:              9
CPU MHz:               1214.062
CPU max MHz:           3100,0000
CPU min MHz:           1200,0000
BogoMIPS:              4190.49
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              6144K
NUMA node0 CPU(s):     0-7
joao@joao-Inspiron-7520:~$ sudo free -m
             total       used       free     shared    buffers     cached
Mem:          7887       3755       4131        600        127       1749
-/+ buffers/cache:       1878       6008
Swap:         8092          0       8092
joao@joao-Inspiron-7520:~$ sudo lsblk
NAME   MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
sda      8:0    0 111,8G  0 disk
├─sda1   8:1    0   100M  0 part
├─sda2   8:2    0  80,5G  0 part
├─sda3   8:3    0     1K  0 part
├─sda5   8:5    0  23,4G  0 part /
└─sda6   8:6    0   7,9G  0 part [SWAP]
sdb      8:16   0  29,8G  0 disk
├─sdb1   8:17   0     8G  0 part
├─sdb2   8:18   0   100M  0 part
└─sdb3   8:19   0  21,7G  0 part
sr0     11:0    1  1024M  0 rom
joao@joao-Inspiron-7520:~$
```

- Ambiente de teste 2
    - Sistema Operacional e distribuição

```
asjacobs@asjacobs-PC:~$ uname -or
3.16.0-38-generic GNU/Linux
asjacobs@asjacobs-PC:~$ lsb_release -irc
Distributor ID: Ubuntu
Release:        14.04
Codename:       trusty
asjacobs@asjacobs-PC:~$
```

- Compilador utilizado

```
ii  gcc                                                    4:4.8.2-1ubuntu6
4         GNU C compiler
ii  gcc-4.8                                                4.8.2-19ubuntu1
4         GNU C compiler
```

- Configurações da máquina

```
asjacobs@asjacobs-PC:~$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    2
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 60
Stepping:              3
CPU MHz:               2300.089
BogoMIPS:              4589.38
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              6144K
NUMA node0 CPU(s):     0-7
asjacobs@asjacobs-PC:~$ sudo free -m
[sudo] password for asjacobs:
             total       used       free     shared    buffers     cached
Mem:          7907       5130       2777        883        255       2547
-/+ buffers/cache:       2327       5580
Swap:         7085          0       7085
asjacobs@asjacobs-PC:~$
```

2 EXPLICAÇÕES E JUSTIFICATIVAS DO CÓDIGO

a) Como foi implementada a concorrência no servidor?

A concorrência dentro do servidor foi estabelecida através da criação de uma thread diferente para cada cliente. O trecho de código que demonstra como foi realizado esse processo é o seguinte:

```
while (1)
{
    client_t *user = malloc(sizeof(client_t));
    user->thread = malloc(sizeof(pthread_t));
    user->socket_addr = malloc(sizeof(struct sockaddr_in));
    user->socket_len = sizeof(struct sockaddr_in);

    puts("Waiting client...");
    if ((user->socket = accept(sockfd, (struct sockaddr*)user->socket_addr, &user->socket_len)) == -1)
    {
        puts("There was an Error accepting connection.");
        exit(-1);
    }
    puts("Client connected!");

    pthread_mutex_lock(&client_m);
    {
        user->next = users;
        users = user;
        bzero(buffer, DEFAULT_BUFFER_SIZE);
        read(user->socket, buffer, DEFAULT_BUFFER_SIZE);
        user->name = malloc(strlen(buffer) + 1);
        user->color = user_colors[current_color];
        current_color = (current_color + 1) % NUM_COLORS;
        strcpy(user->name, buffer);
    }
    pthread_mutex_unlock(&client_m);

    join_room(user, MAIN_ROOM);
    pthread_create(user->thread, NULL, client, (void*)user);
}
```

No caso, é criada uma estrutura diferente para cada cliente, e quando um cliente se conecta ao servidor, a estrutura desse cliente recebe as devidas atribuições e então é colocada juntamente a lista de todos os usuários do chat. A concorrência propriamente dita é estabelecida através da função *pthread_create* que executa a função cliente, sendo esta função responsável por identificar e executar todos os comandos solicitados pelos clientes.

b) Quais estruturas de dados são mantidas pelo servidor com informações dos clientes?

São mantidos dois tipos de estruturas de dados pelo servidor:

- Lista encadeada com estruturas do tipo *client_t*, nos quais alocam o nome do cliente, a cor designada a ele, a sala no qual ele se encontra, o próximo cliente da lista encadeada, além da thread designada para exercer as funções para aquele cliente.

```c
typedef struct client_t
{
    char *name;
    char *color;
    struct room_t *room;
    struct client_t *next;
    int socket;
    int socket_len;
    struct sockaddr_in *socket_addr;
    pthread_t *thread;
} client_t;
```

- Lista encadeada com estruturas do tipo *room_t*, nos quais alocam o nome da sala, o número de usuários contidos nela, o número máximo de usuários, uma *string* que aloca o nome do usuário que recebeu a chamada para *private*, uma estrutura que contém o cliente que realizou a solicitação de *private*, além de um ponteiro para próxima sala da lista, e um ponteiro com o endereço da lista encadeada de usuário

```c
typedef struct room_t
{
    char *name;
    int n_users;
    int max_users;
    char *pvt_receiver_nick;
    struct client_t *pvt_sender;
    struct room_t *next;
    struct client_t **users;
} room_t;
```

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

Sincronizações em acesso aos dados foram utilizados em grande parte do código, todos eles são mostrados abaixo, juntamente com a justificativa para necessidade dos mesmos:

```c
pthread_mutex_lock(&client_m);
{
    for (i = 0; i < room->max_users; i++)
    {
        if (room->users[i] != NULL)
        {
            bzero(buffer, DEFAULT_BUFFER_SIZE);
            snprintf(buffer, DEFAULT_BUFFER_SIZE, "%s[Administrator@%s]: %s%s", color_red, room->name, msg, room->users[i]->color);
            write(room->users[i]->socket, buffer, sizeof(buffer));
        }
    }
}
pthread_mutex_unlock(&client_m);
```

```c
pthread_mutex_lock(&client_m);
{
    for (i = 0; i < room->max_users; i++)
    {
        if (room->users[i] != NULL && room->users[i] != user)
        {
            bzero(buffer, DEFAULT_BUFFER_SIZE);
            snprintf(buffer, DEFAULT_BUFFER_SIZE, "%s[%s@%s]: %s%s", user->color, user->name, room->name, msg, room->users[i]->color);
            write(room->users[i]->socket, buffer, sizeof(buffer));
        }
    }
}
pthread_mutex_unlock(&client_m);
```

```c
pthread_mutex_lock(&client_m);
{
    for (i = 0; i < room->max_users; i++)
    {
        if (room->users[i] != NULL && room->users[i] != user && (strcmp(room->users[i]->name, nick) == 0))
        {
            bzero(buffer, DEFAULT_BUFFER_SIZE);
            snprintf(buffer, DEFAULT_BUFFER_SIZE, "%s[%s->%s@%s]: %s%s", user->color, user->name,  room->users[i]->name, room->name, msg, room->users[i]->colo
            write(room->users[i]->socket, buffer, sizeof(buffer));
        }
    }
}
pthread_mutex_unlock(&client_m);
```

Os trechos de código acima são referentes, respectivamente, as funções *reply_all*, *send_message_to_all* e *send_message_to_user*. O uso das primitivas *pthread_mutex_lock* e *pthread_mutex_unlock* é justificada tendo em vista que deseja-se que as mensagens sejam entregas de maneira ordenada com relação ao tempo (ou seja, o cliente que enviou uma mensagem antes do que outro cliente terá sua mensagem disponibilizada a todos do chat antes que a primeira mensagem enviada por este outro cliente seja emitida).

```c
pthread_mutex_lock(&room_m);
{
    for (it = rooms; it != NULL; it = it->next)
    {
        offset += sprintf(buffer + offset, "%s [%d/%d]\n", it->name, it->n_users, it->max_users);
    }
}
pthread_mutex_unlock(&room_m);
```

O trecho de código acima é referente a função *list_rooms*, o uso das primitivas *pthread_mutex_lock e pthread_mutex_unlock* é justificado tendo em vista que é indesejado a criação ou destruição de salas concorrentemente as listagens das mesmas, ocasionando dessa maneira

inconsistência na apresentação destas na listagem (o cliente pode listar a procura de uma sala, entretanto essa só ser criada durante o processo de listagem ou o cliente pode encontrar uma sala desejada, mas esta ter sido deletada durante o processo de listagem). Teoricamente, a necessidade da atomicidade nesse trecho está ligada ao fato da lista encadeada *rooms* não satisfazer a propriedade do *at-most-once* como estudado em aula.

```
pthread_mutex_lock(&client_m);
{
    for (i = 0; i < room->max_users; i++)
    {
        if (room->users[i] != NULL)
        {
            offset += sprintf(buffer + offset, "[%s%s%s]\n", room->users[i]->color, room->users[i]->name, color_red);
        }
    }
}
pthread_mutex_unlock(&client_m);
```

O trecho de código acima é referente a função *list_users*, o uso das primitivas *pthread_mutex_lock* e *pthread_mutex_unlock* é justificado de maneira semelhante ao da função *list_rooms* demonstrado acima. É indesejada a entrada ou saída de usuários no chat durante a listagem, para que essa não seja gerada de maneira inconsistente (o cliente pode listar a procura de um usuário, entretanto esse só adentrar no chat durante o processo de listagem ou o cliente pode encontrar um usuário, mas esta ter se retirado do chat durante o processo de listagem).Teoricamente, a necessidade da atomicidade nesse trecho está ligada ao fato da lista encadeada *room->users[i]* não satisfazer a propriedade do *at-most-once* como estudado em aula.

```
pthread_mutex_lock(&room_m);
{
    room_t *it;
    for (it = rooms; it != NULL; it = it->next)
    {
        if (strcmp(it->name, room->name) == 0) // Se nome da sala ja existe nao cria
        {
            puts("Room already exists. Please try another name.");
            //room mutex end
            pthread_mutex_unlock(&room_m);
            return;
        }

        if (it->next == NULL) //se fim da lista, cria
        {
            it->next = room;
            //room mutex end
            pthread_mutex_unlock(&room_m);
            return;
        }
    }
    rooms = room; //Se ´e primeiro da lista
}
pthread_mutex_unlock(&room_m);
```

```
pthread_mutex_lock(&room_m);
{
    room_t *it;
    for (it = rooms; it != NULL; it = it->next)
    {
        if (strcmp(it->name, room->name) == 0)
        {
            puts("Room already exists. Please try another name.");
            //room mutex end
            pthread_mutex_unlock(&room_m);
            return;
        }

        if (it->next == NULL)
        {
            it->next = room;
            //room mutex end
            pthread_mutex_unlock(&room_m);
            return;
        }
    }
    rooms = room;
}
pthread_mutex_unlock(&room_m);
```

O trecho de código acima é referente a função *create_room* e *create_pvt_room*, o uso das primitivas *pthread_mutex_lock* e *pthread_mutex_unlock* é justificado de maneira análoga ao da função *list_rooms*. É indesejado que se crie uma sala concorrentemente a uma listagem, pois essa pode se tornar inconsistente. Ademais, a inconsistência pode ser gerada na própria criação da sala, caso uma sala de mesmo nome for criada ou deletada concorrentemente (no caso, poderia ocorrer a situação de salas do mesmo nome serem criadas, causando uma situação não-determinística quando executada a função *join*, por exemplo, ou a situação de uma sala não ser criada por conta de outra sala de mesmo nome, entretanto essa outra sala estar sendo deletada concorrentemente, fazendo que a criação da anterior tenha sido cancelada sem justificativa). Teoricamente, a necessidade da atomicidade nesse trecho está ligada ao fato da lista encadeada *rooms* não satisfazer a propriedade do *at-most-once* como estudado em aula.

```
pthread_mutex_lock(&room_m);
{
    for (it = rooms; it != NULL; it = it->next)
    {
        if (strcmp(it->name, name) == 0)
        {
            if (it->pvt_sender == NULL && it->pvt_receiver_nick == NULL)
            {
                if (it->n_users < it->max_users) // Se cabe mais cliente
                {
                    room_to_join = it; //gets room
                    break;
                }
                else
                {
                    reply(user, "Room is full. Please try later.");
                    pthread_mutex_unlock(&room_m);
                    return;
                }
            }
            else
            {
                reply(user, "Sorry, but that is a private room."); //Se f
                pthread_mutex_unlock(&room_m);
                return;
            }
        }
    }
}
pthread_mutex_unlock(&room_m);
```

```
pthread_mutex_lock(&room_m);
{
    int i;
    for (i = 0; i < room_to_join->max_users; i++)
    {
        if (room_to_join->users[i] == NULL) //Poe
        {
            room_to_join->users[i] = user;
            user->room = room_to_join;
            room_to_join->n_users++;
            break;
        }
    }
}
pthread_mutex_unlock(&room_m);
```

Os trechos de código acima são referentes a função *join_room*, o uso das primitivas *pthread_mutex_lock* e *pthread_mutex_unlock* é justificado visto que é indesejado, por exemplo, que dois *joins* sejam realizados concorrentemente, uma vez que é necessário a verificação do número de*"max_users* em relação a *n_users.* Podemos exemplificar a situação imaginando que um cliente verifica que o número de usuários atuais na sala é menor que o número máximo, entretanto, ocorresse a preempção da thread deste cliente e a thread de outro cliente faz a mesma coisa que a anterior. Como o número atual de usuários ainda não foi atualizado, ambos os clientes entrarão na sala, e no caso, sendo essa sala atualmente com *"n_user = max_users – 1",* geraria o caso do número de usuários exceder o número máximo da sala. Todos os casos anteriores e posteriores nos quais utilizam alguma variável compartilhada da lista encadeada rooms poderiam de alguma maneira causar uma inconsistência, visto que não satisfazem as propriedades do at-most-once como estudado em aula.

```
pthread_mutex_lock(&room_m);
{
    for (it = rooms; it != NULL; it = it->next) // Compara de sala em
    {
        if (it->pvt_receiver_nick != NULL)
        {
            if (strcmp(it->pvt_receiver_nick, user_rcvr->name) == 0)
            {
                user_sndr = it->pvt_sender; //gets sender

                it->users[0] = user_rcvr;
                user_rcvr->room = it;
                it->n_users++;

                break;
            }
        }
    }
}
pthread_mutex_unlock(&room_m);
```

```
pthread_mutex_lock(&room_m);
{
    for (it = rooms; it != NULL; it = it->next)
    {
        if (it->pvt_receiver_nick != NULL)
        {
            if (strcmp(it->pvt_receiver_nick, user_rcvr->name) == 0)
            {
                it->users[1] = user_sndr;
                user_sndr->room = it;
                it->n_users++;

                break;
            }
        }
    }
}
pthread_mutex_unlock(&room_m);
```

Os trechos de código acima são referentes a função *join_pvt_room*, o uso das primitivas *pthread_mutex_lock* e *pthread_mutex_unlock* é justificado pela utilização da variável *rooms,* no caso, poderiam ocorrer casos anteriormente mencionados do tipo exclusão da sala enquanto a lista de salas a procura do *receiver* da chamada *pvt* estivesse sendo analisada.

```
pthread_mutex_lock(&room_m);
{
    if (rooms == room)
    {
        rooms = room->next;
    }
    else
    {
        room_t *it;
        for (it = rooms; it->next != NULL; it->next)
        {
            if (it->next == room)
            {
                it->next = it->next->next;
                break;
            }
        }
    }
}
pthread_mutex_unlock(&room_m);
```

O trecho de código acima é referente a função *delete_room*, o uso das primitivas *pthread_mutex_lock* e *pthread_mutex_unlock* é justificado novamente pela utilização da variável *rooms*, os quais são compartilhados entre várias threads e devem ser gerenciados de maneira atômica afim de eliminar possíveis inconsistências de dados, por exemplo, sabendo que essa função é chamada quando o último cliente de uma sala sai dela, ocorrer de um outro cliente estar adentrando a sala simultaneamente, sendo esta deletada antes que este cliente possa ser inserido.

```
pthread_mutex_lock(&room_m);
{
    for (it = rooms; it != NULL; it = it->next)
    {
        if (it->pvt_receiver_nick != NULL || it->pvt_sender != NULL)
        {
            if (!(strcmp(it->pvt_receiver_nick, user_rcvr->name)) || !(strcmp(it->pvt_sender->name, user_rcvr->name)))
            {
                room_to_delete = it; //gets room
                break;
            }
        }
    }
}
pthread_mutex_unlock(&room_m);
```

O trecho de código acima é referente a função *delete_pvt_room*, o uso das primitivas *pthread_mutex_lock* e *pthread_mutex_unlock* é justificado pela utilização da variável *rooms*, e um caso onde a não-atomicidade da execução do trecho de código geraria inconsistência é análogo ao que vimos na função *delete_room*.

```
pthread_mutex_lock(&client_m);
{
    if (users == user)
    {
        users = user->next;
    }
    else
    {
        client_t *it;
        for (it = users; it->next != NULL; it->next)
        {
            if (it->next == user)
            {
                it->next = it->next->next;
                break;
            }
        }
    }
    close(user->socket);
}
pthread_mutex_unlock(&client_m);
pthread_exit(0);
```

O trecho de código acima é referente a função *exit_chat*, o uso das primitivas *pthread_mutex_lock* e *pthread_mutex_unlock* é justificado através da modificação que ela gera na variável *users*. Alguns casos em que a não-atomicidade poderiam ocasionar inconsistência seria a execução desse comando concorrentemente a qualquer outro comando que venha a mencionar o cliente em questão, como por exemplo listagem de usuários, mensagem privada e requisição para sala privada. Se as operações detectarem que o cliente existe e logo essas operações são validas, e após uma preempção esse

cliente em questão termina sua sessão, ocorrerão problemas devido as posteriores execuções relacionadas a esse cliente não mais existente no sistema.

```
pthread_mutex_lock(&room_m);
{
    for (it = rooms; it != NULL; it = it->next)
    {
        if (it->pvt_receiver_nick != NULL)
        {
            if (strcmp(nick, it->pvt_receiver_nick) == 0)
            {
                requested = 1;
            }
        }
    }
}
pthread_mutex_unlock(&room_m);
```

O trecho de código acima é referente a função *was_user_pvt_requested*, o uso das primitivas *pthread_mutex_lock* e *pthread_mutex_unlock* é justificado pela utilização da variável *rooms*. Uma situação que a não-atomicidade do trecho de código poderia gerar inconsistência seria no caso da lista ser percorrida concorrentemente a execução de uma operação de criação de sala por parte do cliente que emitiu o convite de chamada privada, e sendo assim, o mesmo sairia da sala ocasionando sua remoção da lista (no caso, restariam 0 usuários na sala).

d) Funcionalidades adicionais implementadas?

Foram implementadas três novas funcionalidades no programa, as quais não eram especificadas no enunciado do trabalho.

- Cada usuário possui uma cor própria, utilizado para melhor distinção de usuário para usuário em salas com grande número de pessoas. A cor do usuário é definida pelo programa logo na sua inicialização.
- Foi implementado a função de mensagem particular. Um usuário que deseja enviar uma mensagem a outro sem que essa mensagem seja visualizado pelos usuários que participam do grupo pode utilizar o comando /<nick>, sendo nick o nome atribuído ao usuário, seguido da mensagem.
- Foi implementada a função *private call* (/pvt <nick>). Essa função possibilita que o usuário se comunique com outro em particular através de uma sala reservada, onde somente estes dois terão acesso. A linha de execução do comando começa com um dos clientes solicitando uma chamada privada a outro, caso aceito, os dois são redirecionados para uma sala privada, caso recusado, a sala não é criada.

3 PRINCIPAIS DIFICULDADES ENCONTRADAS

Dentre as dificuldades enfrentadas durante o desenvolvimento do código, dois deles devem ser ressaltados. Primeiramente, a sincronização entre os diferentes clientes, que quando não sincronizados

corretamente, geravam *deadlocks*, tomando algum tempo para serem resolvidos um a um. Em segundo lugar, o comando */pvt* foi de complexidade superior a todas as outras, visto o número de verificações necessárias para que ela pudesse ser estabelecida. Para resolução da mesma foi necessário a criação de campos nas salas que identificassem e validassem a existência de um *sender* e *reciever*, e por conseguinte, executar as ações posteriores sem maiores problemas.