Relatório
Trabalho 1

Universidade Federal de Santa Catarina

8 de Agosto de 2021

Samuel Cardoso
19100544

Nicolas Vanz
19100541

Implementação do projeto

Para nossa implementação desse projeto, usamos 3 arquivos. Um *common.h* contendo implementações comuns que seriam usadas tanto no cliente quanto no server, como por exemplo defines e structs que definem os padrões de mensagem que serão trocados entre servidor-cliente ou entre cliente-cliente. Um arquivo *server.c* e um *client.c* onde pode-se encontrar o ciclo de vida de ambos os processos e suas devidas funções para quem cumpram seus propósitos.

Common

No arquivo *common.h* teremos uma principal função que é a de definir a estrutura pela qual serão trocadas informações entre os nodos dessa "rede" de comunicação, os defines se tornam complementos para esta estrutura já que eles vão, tirando alguma exeções, ser opcodes de mensagens a serem enviadas:

```
1   /* Requests */
2   #define OP_FIND_MATCH          0
3   #define OP_YOUR_TURN           1
4   #define OP_SCORE               2
5   #define OP_FINISHED_MATCH      3
6   #define OP_REGISTER            4
7   #define OP_DISCONNECT          5
8   #define OP_AMICONNECTED        6
9
10  /* Replies */
11  #define OP_CONNECT_TO      7
12  #define OP_WAIT_FOR_RIVAL  8
13  #define OP_SUCCESS         9
14  #define OP_FAIL            10
15
16  /* error codes */
17  #define NAME_IN_USE        1
18  #define ALREADY_REGISTERED 2
19  #define NOT_REGISTERED     3
20  #define ALREADY_IN_GAME    4
21  #define NOT_IN_GAME        5
```

Server

O servidor usa a chamada poll() para gerenciar as conexões de clientes. Quando uma mensagem é enviada para o servidor, este a trata através da função handle_pollin(), que identifica o propósito da mensagem (nova conexão, requisição de operação ou desconexão) e a trata. As operações que o servidor realiza são: computar pontos ao término de uma partida, enviar a pontuação de cada jogador, conectar dois jogadores para uma partida, registrar um novo jogador e verificar se um jogador está conectado. Destaca-se que um jogador pode ser identificado por nome ou ip e porta, dependendo do interesse. As principais estruturas que o servidor tem são: uma lista de conexões, uma lista de jogadores e uma lista de jogos em andamento. Com elas o servidor gerencia todas as operações que ele é responsável. Para juntar dois jogadores em uma partida, o servidor guarda em uma estrutura a referência ao primeiro jogador que requisitou a partida e quando um segundo jogador requisita uma partida também, o servidor envia mensagens para os dois, uma requisitando que o cliente se conecte com seu rival e outra requisitando que o cliente espere a conexão do seu rival, assim o jogo é controlado pelos clientes e apenas no final do jogo o servidor é contactado, sendo informado do resultado final (quem ganhou ou se houve empate).

Cliente

No Cliente teremos 2 implementações, a implementação do Jogo da Velha e a implementação do Cliente em si, por isso para uma melhor explicação do Cliente iremos explicar as funções principais.

O *main()* irá diretamente tenta conectar ao servidor, servidor este que deve estar previamente definido no código em formato de ipv6.

```
#define SERVER_IPV6 <your_ip>
```

A função *startclient()* tem o papel de definir o socket para comunicação entre o cliente-servidor e retornar o mesmo para uso posterior.

Teremos o *mainloop()* onde o Cliente permanecerá após ter definido seu socket com o servidor e enquanto não estiver em um jogo, neste estado o cliente pode requistar o score, tentar se registrar com o servidor e pedir por um jogo.

O cliente também conta com as funções *inscribe()* e *request_match()* que serão responsáveis por registrar o cliente com o servidor e requisitar um jogo ao servidor respectivamente.

Já as funções *wait_rival()* e *connect_to_rival()* como o nome diz, serão as funções chamadas após um requisição ao servidor por um jogo, o servidor então irá definir qual dos jogadores deve fazer o papel de "servidor" e qual deve tentar se conectar, está conexão entre clientes se dará pela porta 4000. Ambas chamarão a função *match()* que será responsável por mostrar e administrar o jogo em ambos os Clientes.

A função *send_next_round()* é responsável por cada interação de um dos Clientes, ela que irá verificar se as entradas do usuário são válidas e chamar funções auxiliares para o mesmo propósito.

As demais funções são funções que tem o papel mostrar na tela os resultados ou de verificar e setar valores das matrizes para os Clientes, seja valores que o Cliente escolheu e enviou para o outro Cliente, ou os valores de um Cliente recebe. Vale lembrar que as verificações de disponibilidade são feitas localmente em ambos os clientes então um mensagem só é enviada para outro cliente caso o valor seja válido ou se o jogo tiver terminado por vitória ou por todos os campos estarem cheios.

Limitações

Como o POLL criado no projeto é definido por um define, ele é estático, o que torna o atual projeto não preparado para aumentar o número de requisições dinamicamente. Temos também um problema no caso de uma queda de um dos cliente acontecer muito rapidamente após um match, se no exato momento do match um dos clientes cair o outro ficará preso no *accept()* ou em um loop para a tentativa de conexão com o outro cliente. Também temos uma quantidade limitada de jogadores totais, além de serem com dados voláteis e não permanetes como seria o ideal. Como o servidor não é distribuído, é indiferente em qual servidor um Cliente deve se conectar, porém um Cliente de um servidor não poderá jogar com o Cliente de outro servidor.

Realizamos a tentativa de tornar o programa funcional em sockets passando pelo protocolo TCP (uso de ipv6), porém encontramos dificuldade na realização da serialização das estruturas de mensagem enviadas. Localmente não existe este problema já que as mensagens não passam pela serialização e o peso da estrutura permance o mesmo sempre, porém com a serialização, o peso é variável, o que nos levava a receber mensagens com valores trocados ou totalmente distintos do objetivo. Abaixo vamos deixar um pouco desta tentativa de serialização para envio AF_INET6.

```
void deserialize(struct message *msg, char *buf) {
    sscanf(buf, "%d %d %s %d %d %c %d %s %d %d",
        &msg->opcode,
        &msg->body.finish.points_to_add,
        msg->body.inscribe.name,
        &msg->body.nextround.col,
        &msg->body.nextround.row,
        &msg->body.nextround.value,
        &msg->body.nextround.won,
```

```
        msg->body.reply.ip,
        &msg->body.reply.port_nr,
        &msg->body.reply.error_code
    );
}
```

```
void serialize(struct message *msg, char *buf) {
    sprintf(buf, "%d %d %s %d %d %c %d %s %d %d",
        msg->opcode,
        msg->body.finish.points_to_add,
        msg->body.inscribe.name,
        msg->body.nextround.col,
        msg->body.nextround.row,
        msg->body.nextround.value,
        msg->body.nextround.won,
        msg->body.reply.ip,
        msg->body.reply.port_nr,
        msg->body.reply.error_code
    );
}
```

Até conseguimos realizar a troca de mensagens entre cliente-cliente e servidor-cliente, porém em algumas partes da estrutura com a de score, os valores ainda não estavam vindo corretamente. Abaixo uma representação do uso desta serialização:

```
if (score_board_lookup_ip(ip, port) >= 0) {
        response.body.reply.error_code = (-ALREADY_REGISTERED);
    } else if (score_board_lookup_name(name) >= 0) {
        response.body.reply.error_code = (-NAME_IN_USE);
    } else {
        for (int i = 0; i < MAX_PLAYERS; i++) {
            if (score_board.scores[i].points == POINTS_NULL) {
                strcpy(score_board.scores[i].name, name);
                score_board.scores[i].points = 0;
                strcpy(score_board.scores[i].ip, ip);
                score_board.scores[i].port = port;

                score_board.nplayers++;
                response.opcode = OP_SUCCESS;

                printf("New player registered: %s\n", name);
                break;
            }
        }
    }
}
```

```
init_message(&request);

```

```
3       request.opcode = OP_FIND_MATCH;
4       serialize(&request, buf);
5       send(fd, buf, strlen(buf) * sizeof(char), 0);
6       recv(fd, buf, BUFFER_LENGTH*sizeof(char), 0);
7       deserialize(&request, buf);
8
9       if (request.opcode == OP_FAIL) {
10          error = -request.body.reply.error_code;
```