Instituto Superior Técnico

MEEC

Programação de Sistemas

Projeto

Multiplayer Pacman

Alexandre Rodrigues, 90002

Índice

1.1 Arquitetura

1.1.1 Nós

Este projeto implementa uma arquitetura servidor/cliente simples. O servidor aguarda a conexão dos clientes, comunicando com estes de modo a receber e aplicar o *input* do jogador, e enviar o resultante estado do jogo.

1.1.2 Módulos

O cliente e o servidor estão ambos divididos em três módulos, sendo estes o módulo principal de jogo, o módulo de mensagem e o de conexão.

O primeiro dos três implementa a lógica que rege o jogo *Pacman*, como as interações entre os personagens e *delays* nas ações.

O módulo de mensagem implementa funções que definem o protocolo de comunicação entre o cliente e o servidor. Por exemplo, enquanto o módulo de mensagem do servidor implementa a função *message_send_board*, o módulo do cliente implementa a função análoga *message_recv_board*.

Por último, o módulo de conexão lida, como seria de esperar, com a conexão entre o cliente e o servidor. No cliente, por exemplo, implementa a função que faz a ligação inicial ao servidor e a que recebe as mensagens do servidor. No lado do servidor trata ainda de guardar informação relevante aos clientes atualmente conectados.

1.1.3 Threads

Enquanto que o cliente tem apenas duas *threads*, o servidor tem uma *thread* principal e tantas outras quantos os clientes que estiverem conectados.

O cliente tem uma *thread* responsável por desenhar a *board* e o estado do jogo em geral, bem como adquirir o *input* do jogador e enviá-lo para o servidor, e uma *thread* responsável por receber mensagens do servidor (*recv_from_server*), atualizando o estado do jogo comforme o necessário.

A *thread* principal do servidor processa o jogo em si, alterando o estado deste consoante os movimentos enviados pelos clientes, e enviando o resultado para os clientes. Cada cliente que se conecta causa a criação de uma *thread* que recebe exclusivamente as mensagens do respetivo cliente (*recv_from_client*), atualizando a informação utilizada pela *thread* principal.

1.2 Organização do código

O código está dividido em três categorias principais, sendo estas código que diz respeito ao cliente, ao servidor, e a ambos. A diretoria do projeto está organizada da forma seguinte:

```
Project
├── bin
├── board-drawer
├── res
└── src
    ├── common
    │   ├── message.c
    │   ├── message.h. .4 pacman.c
    │   ├── pacman.h
    │   ├── UI_library.c
    │   ├── UI_library.h
    │   ├── utilities.c
    │   ├── utilities.h
    │   ├── vector.c
    │   └── vector.h
    ├── client
    │   ├── client_connection.c
    │   ├── client_connection.h
    │   ├── client_message.c
    │   ├── client_message.h
    │   ├── client.c
    │   └── client.h
    └── server
        ├── server_connection.c
        ├── server_connection.h
        ├── server_message.c
        ├── server_message.h
        ├── server.c
        └── server.h
```

As diretorias *client* e *server* têm o código que diz respeito ao cliente e ao servidor, respetivamente. Cada uma delas implementa os respetivos módulos.

Alguns destes módulos dependem de funções, estruturas ou definições da diretoria *common*. Por exemplo, *message.h* define *MessageTypes*, que os módulos de mensagem utilizam para identificar e processar corretamente mensagens entre eles. *pacman.c* implementa a *board* do jogo, usada igualmente pelo cliente e servidor. *vector.c* e *utilities.c* implementam funções mais genéricas, como operações vetoriais, alocação de memória com verificação, ou conversões de formato.

O código está comentado nos ficheiros *header*, cada função sendo sucintamente explicada pelo comentário que a antecede. Os ficheiros de código também têm comentários ao longo das funções e estruturas quando código não é evidente. De qualquer dos modos, o código foi escrito de modo explicito e verboso, quer seja no nome das funções ou variáveis, com o intuito de facilitar a sua leitura.

1.3 Estruturas de dados

O projeto possui algumas estruturas de dados comuns a ambos os lados servidor e cliente, bem como algumas que são análogas mas diferentes, devido aos requesitos diferentes do cliente e do servidor. Por exemplo, observando a estrutura *Player*, que guarda informação relevante a um jogador no contexto do jogo, do lado do cliente:

```
typedef struct _Player
{
```

```
    unsigned int player_id;
    Color color;
    unsigned int score;
    int powered_up;
    Vector* pacman_pos;
    Vector* monster_pos;
} Player;
```

Observamos que esta é diferente da estrutura análoga do lado do servidor:

```
typedef struct _Player
{
    unsigned int player_id;
    Color color;
    int powered_up;                     // Stores 0, 1 or 2, depending on how
        many monsters the pacman can still eat
    unsigned int score;                 // The number of things eaten
    Vector* pacman_pos;
    char pacman_move_dir;               // Movement direction
    struct timespec pacman_last_move_time; // Time of last movement
    Vector* monster_pos;
    char monster_move_dir;
    struct timespec monster_last_move_time;
} Player;
```

Como seria de esperar, o servidor tem uma implementação mais complexa da mesma estrutura, pois é este que é responsável pelo processamento dos dados. Guarda a mais a direção de movimento do jogador, bem como o tempo em que o último movimento foi executado, para ambos os personagens.

A estrutura *Game* é o método principal de passar informação às diversas funções (de modo a evitar variáveis globais), contendo toda a informação que diz respeito ao jogo. Mais uma vez existem discrepâncias na sua implementação dado que no lado do cliente:

```
typedef struct _Game
{
    int server_socket;
    unsigned int player_id;
    Board* board;
    unsigned int n_players;
    Player** players;
    unsigned int n_fruits;
    Fruit* fruits;
} Game;
```

E no lado do servidor:

```
typedef struct _Game
{
    Board* board;
    unsigned int max_players;
    unsigned int n_players;
    Player** players;
    unsigned int n_fruits;
    Fruit** fruits;
} Game;
```

O cliente necessita de saber o seu *player_id* e, por conveniência, dado que o cliente só comunica

com uma outra entidade, a socket do servidor. Já o servidor guarda também o número máximo de jogadores, que o cliente não precisa de saber.

Quanto às frutas, estas são implementadas *client-side* por:

```
typedef struct _Fruit
{
    unsigned int fruit_type; // Cherry or lemon
    int is_alive;            // Is it on the board? (Or waiting to respawn)
    Vector* pos;
} Fruit;
```

E do lado do servidor:

```
typedef struct _Fruit
{
    unsigned int fruit_type; // Cherry or lemon
    int is_alive;            // Is it on the board? (Or waiting to respawn)
    struct timespec eaten_time; // Time it was last eaten
    Vector* pos;
} Fruit;
```

Mais uma vez, como o servidor implementa o jogo em si, precisa de guardar o instante em que a fruta é comida para fazer o seu *respawn*, algo que não acontece no cliente.

A *board* do jogo é implementada de forma sucinta e privada:

```
// The game board
typedef struct _Board
{
    Vector* board_size;
    unsigned int** board;
} Board;
```

Por fim, os vetores são também definidos, simples e privadamente por:

```
typedef struct _Vec
{
    int x;
    int y;
} Vector;
```

1.4 Protocolo de comunicação

A comunicação entre o cliente e o servidor é feita através de mensagens discretas, identificadas pelo seu *MessageType*:

```
typedef uint16_t MessageType;
enum _MessageType {
    // Terminates a message, used for checking allignment
    MESSAGE_TERMINATOR = UINT16_MAX,
    // Player color
    MESSAGE_COLOR = 0,
    // The game board
    MESSAGE_BOARD,
    // ...
```

```
    // Resto das mensagens em common/message.h
    // ...
    // Alerts the server is full
    MESSAGE_SERVER_FULL
    };
```

message.c também implementa diversas funções de envio e recebimento de dados de tipos específicos e explicitos quanto ao tamanho e sinal. Tem-se por exemplo:

```
int message_send_uint16_t(int socket, uint16_t message)
{
    uint16_t m_net = htons(message);
    return send_all(socket, (void*)&m_net, sizeof(uint16_t));
}
int message_recv_uint16_t(int socket, uint16_t* message)
{
    int ret = recv_all(socket, (void*)message, sizeof(uint16_t));
    *message = (uint16_t)ntohs(*message);
    return ret;
}
```

A partir destas funções, são implementadas funções mais abstratas nos módulos de comunicação específicos ao cliente ou servidor. Tomando o exemplo simples (o qual é seguido pela maioria das funções do mesmo tipo) da função (do lado do servidor, em *server_message.c*) que envia o *player_id* ao seu cliente respetivo:

```
void message_send_player_id(int socket, unsigned int player_id)
{
    message_send_uint16_t(socket, (uint16_t)MESSAGE_PLAYER_ID);
    message_send_uint32_t(socket, (uint32_t)player_id);
    message_send_uint16_t(socket, (uint16_t)MESSAGE_TERMINATOR);
}
```

E a sua análoga encarregue do recebimento (no cliente, em *client_message.c*):

```
void message_recv_player_id(int socket, unsigned int* player_id)
{
    message_recv_uint32_t(socket, (uint32_t*)player_id);
}
```

Como se observa, a função de envio envia não só os dados. Envia primeiro o *MessageType*, de modo a que o cliente possa identificar o conteúdo, seguido dos dados da mensagem, e finalmente um terminador de mensagem. Este último permite já fazer uma validação preliminar dos dados quanto ao seu comprimento como se poderá observar de seguida.

A função de recebimento está encarregue exclusivamente de receber os dados. Isto porque a identificação e verificação do alinhamento da mensagem é feito em *recv_from_server* (em *client_connection.c*), uma versão resumida da qual está abaixo:

```
void* recv_from_server(void* _game)
{
    // Cast to game
    Game* game = (Game*)_game;
    int server_socket = game_get_server_socket(game);

    // Probe for new messages
    while (1)
```

```
    {
        // Determine message type
        MessageType mt;
        int ret = message_recv_uint16_t(server_socket, (uint16_t*)&mt);
        // ...
        switch (mt)
        {
        // ...
        case MESSAGE_PLAYER_ID:
        {
            unsigned int player_id;
            message_recv_player_id(server_socket, &player_id);
            game_set_player_id(game, player_id);
            break;
        }
        // ...
        default:
            break;
        }

        // Receive the terminator
        message_recv_uint16_t(server_socket, (uint16_t*)&mt);
        if (mt != MESSAGE_TERMINATOR)
            message_misaligned();
    }
    client_quit();
    return NULL;
}
```

A função recebe o tipo de mensagem, escolhendo através do *switch* a função adequada para interpretar os dados que se seguem. Depois da mensagem ser lida verifica o alinhamento e, caso não esteja alinhada, o cliente interrompe a execução. Na prática, nunca há um desalinhamento dos dados, sendo que isto foi implementado de modo a ajudar com o densenvolvimento das diversas mensagens.

A função *recv_from_client* do servidor é quase idêntica, bem como todo o processo detalhado anteriormente. No entanto, neste contexto não faria sentido interromper a execução caso exista desalinhamento de mensagens. Se isso fosse feito, qualquer cliente malicioso poderia enviar um pacote de dados demasiado longo ou curto e parar o servidor. Neste caso, um desalinhamento dos dados resulta então na terminação da conexão com o cliente que o causou.

Há que salientar que um desalinhamento nunca ocorre em situações normais, pois as mensagens foram desenhadas de forma a serem discretas e de tamanho explicito, sendo que não há risco do uma das partes ler menos ou mais informação do que o que a mensagem contém.

O servidor tem ainda outra forma de comunicação única, a qual utiliza para atualizar os clientes com o estado do jogo. A função *send_to_all_clients* envia uma mensagem idêntica para todos os clientes atualmente conectados a comando da *thread* principal. Um resumo dessa função encontra-se abaixo:

```
void send_to_all_clients(Game* game, MessageType message_type, void* extra_data)
{
    pthread_mutex_lock(&client_array_lock);
    // For every client
    for (unsigned int i = 0; i < n_clients; ++i)
    {
        // Send the message defined by message_type
        switch (message_type)
        {
```

```
            // ...
            case MESSAGE_PLAYER_LIST:
                message_send_player_list(client_array[i]->socket, game);
                break;
            case MESSAGE_PLAYER_DISCONNECT:
            {
                unsigned int* player_id = (unsigned int*)extra_data;
                message_send_player_disconnect(client_array[i]->socket, *player_id);
                break;
            }
            // ...
            default:
                break;
            }
        }
        pthread_mutex_unlock(&client_array_lock);
    }
```

1.5 Validação de dados

Na transação de informação, é especialmente importante protejer o servidor de informação possívelmente prejudicial. Deste modo, é necessário verificar os dados que chegam ao servidor são válidos, e agir de acordo com o resultado. Um exemplo de verificação de dados está representado abaixo, um excerto da função *receive_from_client* (funcionalmente idêntica à analisada acima) do ficheiro *server_connection.c*:

```
        case MESSAGE_MOVE_PAC:
        {
            char move_dir;
            message_recv_movement(client->socket, (char*)&move_dir);
            if (move_dir == 'w' || move_dir == 'a' || move_dir == 's' || move_dir == 'd'
                || move_dir == (char)0)
                player_set_pac_move_dir(player, move_dir);
            break;
        }
```

Caso um cliente enviasse uma direção de movimento inválido, isso poderia ter consequências críticas para o funcionamento do servidor, a não ser que o movimento também fosse verificado mais à frente. Ao verificar assim que a mensagem é recebida, e ignorando-a caso não seja válida, assegura-se a proteção do servidor de dados inválidos e simplifica-se o processo de *error-checking*.