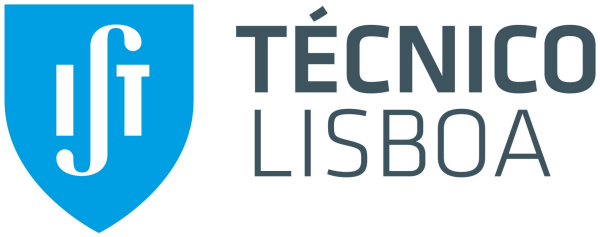
Universidade de Lisboa

Instituto Superior Técnico

Integrated Master's in Aerospace Engineering



Systems Programming

Project Report

Nuno Brandão - 85232

Lisbon

July 2020

Contents

1 Architecture

There were some concerns when developing the code regarding architecture, since different functionalities were implemented at different stages, it was important to maintain a solid structure and to maintain separation of concerns on the system.

Regarding architectural patterns, the adopted **structure** was **layered architecture**. This was clearly the best structure to adopt, since there was a need to group related functionalities together, while keeping a hierarchical organization. This also assured the flexibility and maintainability of the system, while not only enabling testability but also ensuring better performance.

Regarding **deployment**, choosing a semi-**Client/Server** architecture style was fairly trivial. The reason that it is not a pure Client/Server deployment style is because given a Client request, the server only responds if that request is valid, and not only it responds to the client that made the request, but to all other clients as well. There are many reasons, but the fact that it is a multiplayer online game (with more than 2 clients at the same time) requires a network with centralised data access. This of course, allows for better security, since the client and server are separate.

The architecture style employed at the **communication** level was **RPC** - Remote Procedure Call. This was due to the need to enable the Client to make different types of requests to the server, either it be for different inputs, differentiating characters, or even regarding connections and disconnections.

Figure 1.1 shows the system architecture through different layers, including the server and client interactions and behaviour with the shared memory component of the system.

Each layer will be discussed with greater detail in further sections, including the modules from each layer (and their respective functionalities), that is, the structures used to store data and the functions and threads used to handle said data.

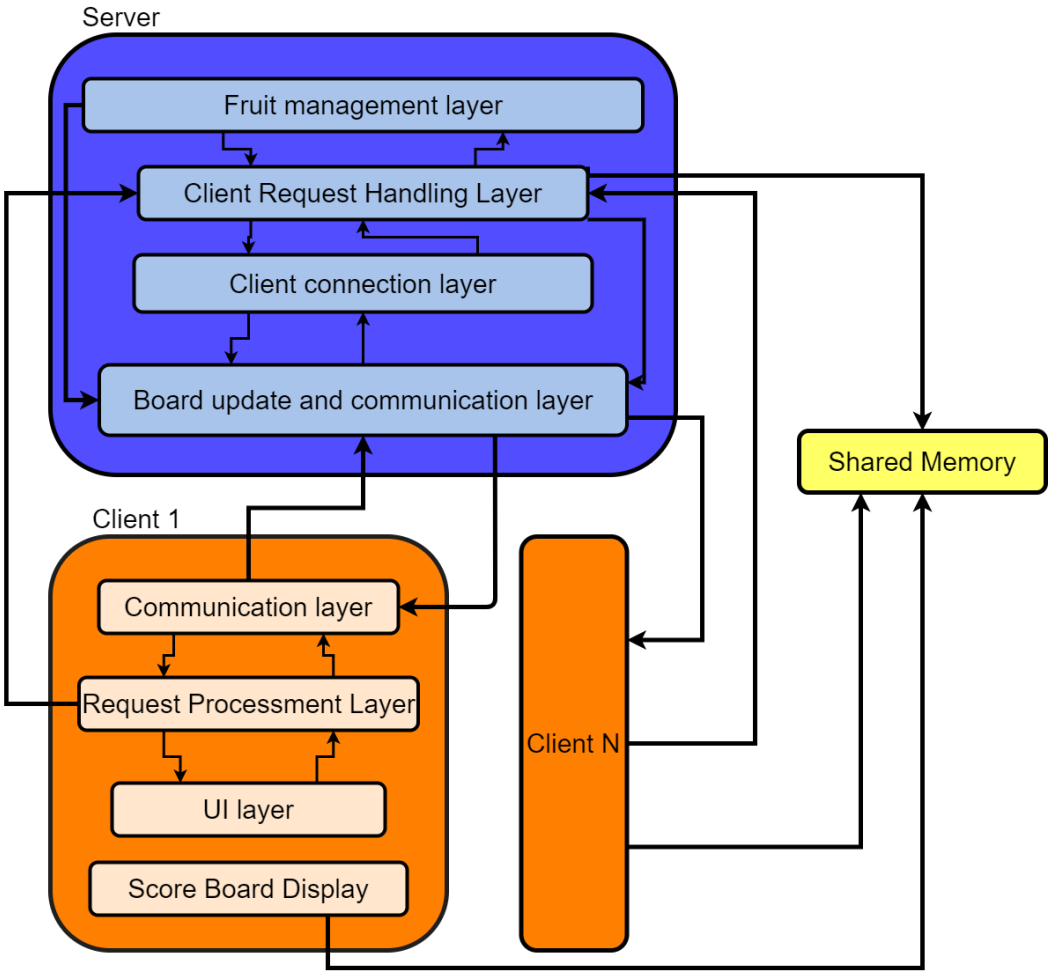


Figure 1.1: Layered Architecture of the System

2 Data Structures

In this section, the data structures used by the server and by the client are presented.

The game board size is stored in variable *BoardSize*[2]. The board is represented on the server by a 2 dimensional array, having in each position the structure:

```
struct BoardInfo{
    int occupied; //0 if unnocupied, 1 if occupied
    char type;  //M monstro, P pac, B tijolo, F fruta
    int ClientNum;  //-1 if Brick, or Fruit
};
```

Variable *occupied* is used to verify if a play is made to an already occupied place, variable *type* indentifies the type that is currently at that board position and variable *ClientNum* identifies the client that owns the character, being -1 if it's an object from the server (brick or fruit).

The information about each client is stored on the server in an array of structures *ClientThreadInfo*:

```
struct ClientThreadInfo{
    int clientActive;           //0 if client has exited, 1 if active
    pthread_t threadID;         //ID of thread treating a certain client
    int clientfd;               //fd to write/read to client
    struct sockaddr_in client_addr;         //Client Address information
    int x_pac, y_pac;                       //Pacman position
    int x_moob, y_moob;                     //Monster position
    int colour[3];                          //Client Colour
    double lastPacPlay, lastMoobPlay;     //instance when last accepted play
    int superPowerEats;      //Number of super Pacman available eats
    int superPower;          //0 if normal PAc, 1 if super power
};
```

The variable *clientActive* is used to insert a new player in a position from the array that isn't currently being occupied by another connected player. The client characters position are stored in the variables x_pac, y_pac, x_moob and y_moob. To implement the time interval between character moves, the server stores the time instance of the last character play in the variables *lastPacPlay* and *lastMoobPlay*. When a pacman eats a fruit it becomes a Super Powered Pacman, storing the information regarding how many eats before returning to normal pacman in variable *superPowerEats*, setting the variable *superPower* to 1 while in super power mode. Variable *threadID* stores the ID of the thread threating the client, and *colour*[3] stores the player colour.

When a new client connects, the server needs to send the initial data for the client to initialize the SDL window, regarding board size, its player number and character's positions, the maximum number of clients allowed and the timer to read the score board from shared memory (to synchronize score board display with the other clients). This information is sent using the following structure:

```
typedef struct messageInit{
  int boardSize_x;
    int boardSize_y;
    int clientNum;
    int pac[2];
    int moob[2];
    int MAXCLIENTS;
    int timeToShowScore;
} messageInit;
```

After the client process the initial message, when a move is made the player sends a request to the server in the following message structure:

```
typedef struct message{ //Messages from client to server
    int num;
    char type;       //P -> pacman, M -> monster, S -> SuperPoweredPac, B -> Brick
    int x, y;   //Position to analyse by server
} message;
```

The server uses this data to validate player number stored in variable *num*, and analyse the requested play based on variables $type, x$ and y. Then, if the play was valid, the server answers to every connected player sending messages of type $ServMsg$.

```
typedef struct ServMsg{ //Messages from server to client
    int num, colateralNum;
    char type, colateralType;
    int x_new, y_new, x_old, y_old;
    int x_colateral, y_colateral, colateral_x_old, colateral_y_old;
    int colateralDamage;    //0 if no colateralDamage, 1 if so
    int paintColour[3], colateralPaintColour[3];
} ServMsg;
```

Clients update the board when receive a message from the server. This happens either updating the clients board initially and when any client makes a play. It contains data about a clients play, regarding what type of character to move, the new and old positions, and the colour to paint it. However, a client play can have effects other character other than its own, and if so it also sends data to paint the character that suffered colateral damage, stored in variables with *colateral* in its name.

To store synchronization variables in the server, a structure named $SyncVars$ was created:

```
struct SyncVars{
    pthread_rwlock_t rwLockActiveClient;
    pthread_mutex_t muxGameRules;
    pthread_rwlock_t rwLockBoard;
    pthread_rwlock_t rwLockClientInfo;
    pthread_mutex_t muxFruits;
    pthread_cond_t condFruits;
}SyncVars;
```

Read/write lock variables:

- *rwLockActiveClient* - used to guarantee that variable *clientActive* in the array of structures $ClientThreadInfo$ is only accessed when no thread is writing to it (guarantees a valid player number assignement).
- *rwLockBoard* - used to guarantee that the board can only be read when no thread is writing to it.
- *rwLockClientInfo* - used to guarantee that the variables x_pac, y_pac, x_moob and y_moob in the array of structures $ClientThreadInfo$ is only accessed when no thread is writing to it.

Mutex variables:

- *muxGameRules* - used to guarantee that only one thread access function *ApplyGameRules*, that process data regarding a players move.
- *muxFruits* - used to allow the condition variable *condFruits* to be set on wait and to be signaled.

Condition variable *condFruits* - used to wait when the number of fruits on board is right, and to be signaled when a fruit is eaten and needs to be respawned.

The score board is saved on the server on an array of structures *sharedScore*:

```
1 //Shared Memory Structure
2 typedef struct sharedScore{
3     int activeClient;
4     int num;
5     int monsterEaten;
6     int superPacEaten;
7     int fruitEaten;
8 } sharedScore;
```

This structured have data regarding number of active players and the score of each player of eaten fruits, pacmans and monsters. This structure is updated every time a play interacts with an object other than a brick or the other character from the same client. Variable *num* identifies client number, *monsterEaten* stores data about how much characters the monster from this client has eaten, *superPacEaten* stores the characters eaten by the super powered pacman, and $fruiEaten$ stores the number of fruits eaten by this client. The setup to the use of this structure in a shared memory is made, and the clients read from it every 1 minute (server time).

3 Code Organization

3.1 Server

3.1.1 File ServThreads.c

This file stores the functions used on threads, namely functions *ClientThreadShow*, *threadAccept* and *ShowFruitsOnBoard*.

3.1.2 File ServLib.c

Every functions that are used on any thread of the server are stored here.

3.1.3 File Serv.c

This is the server main function, and needs to be called with the following input arguments:

- Number of places in x on board.
- Number of places in y on board.

. In order to set the bricks for a new board, when the server is initialized it prompts a new SDL window (in a new process) that allows to place and erase bricks inside the board. When it's done, close the window and the server will read from a file the desired brick position and paint them on the server board. This is done using function *GenerateBoardStructure*:

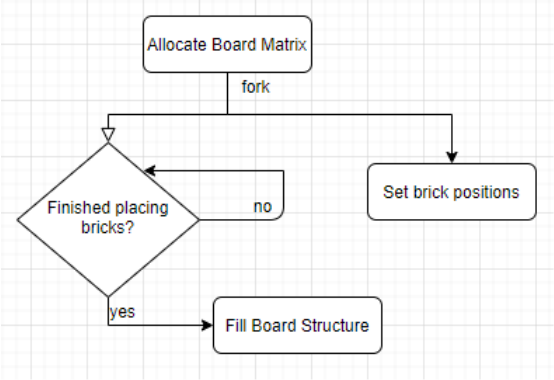


Figure 3.1: *GenerateBoardStructure* fluxogram

Then, the server sets up the stream socket to receive connections (function *servSocketSetup*), initialyze the synchronization variables (function *initializeSyncVars*) and creates the shared memory (function *setupSharedMemory*).

The thread that handle new client connections and manage fruits on board is created, and the main function enters in a cycle of reading SDL events until the program is closed. It has 2 possible SDL events:

- SDL_Quit - Happens when the server is closed.
- Event_Show - Happens every time a player makes a valid play. It receives data regarding characters to paint and where to paint them, including all characters involved in that play.

The fluxogram of the thread that handles client connections (*threadAccept*) is represented in Figure 3.2:

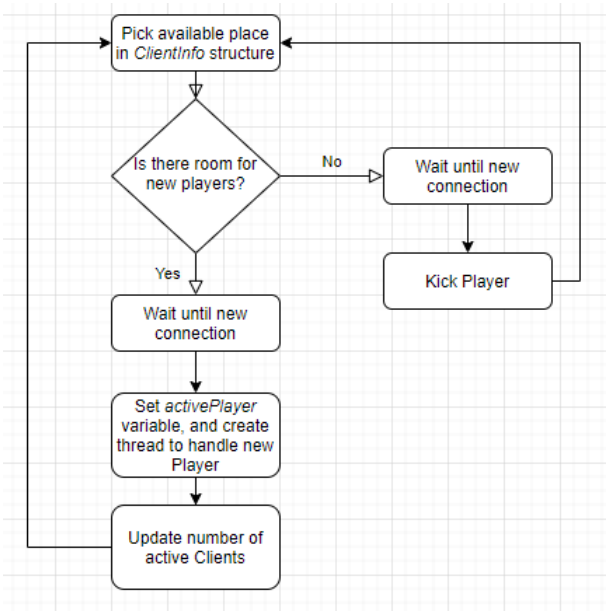


Figure 3.2: *threadAccept* fluxogram

The fluxogram of the thread that handles the fruits on board is represented in the following Figure:

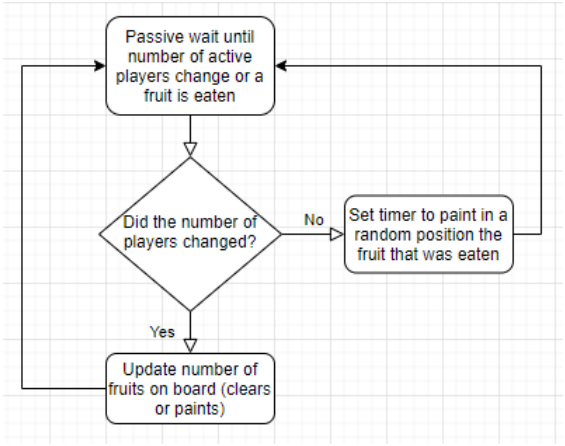


Figure 3.3: *ShowFruitsOnBoard* fluxogram

When a new client connects, if there is space available on the board, *threadAccept* creates a new thread that will handle this client requests. The threads are created to run function *ClientThreadShow*, and its fluxogram is represented in figure 3.4:

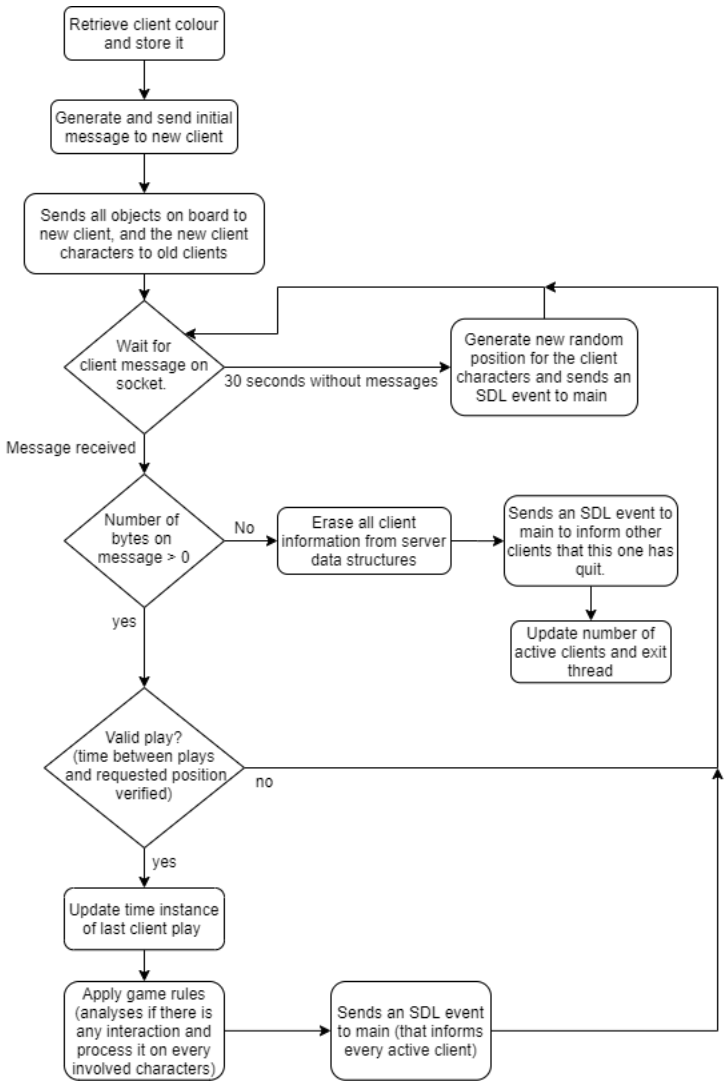


Figure 3.4: *threaddClientShow* fluxogram

3.2 Client

3.2.1 File ClientLib.c

This file contains the functions used on the client (including the functions run by the thread that receives server messages).

3.2.2 File Client.c

This is the client main function, and needs to be called with the following input arguments:

- IP from the server to connect
- Port to connect
- Colour of characters (r, g or b)

. When a new client opens, it initializes function *readFromSharedMemory* to run when a *SIGALRM* signal happens (to read from shared memory), generates its own colour using *GenerateOwnColour* function (creates random colours, emphasising more on the one chosen by the player), setups connection with the server socket with function *SocketSetupAndConect* and initializes the shared memory to read the score board from. Then, it sends its own colour to the server, and receives the initial message from the server.

With the initial information, it stores client number, characters positions and creates the SDL game window. Then, the program creates a thread that runs function *threadShow*, and the main steps into a cycle of reading SDL events. There are 4 possible events:

- SDL_QUIT - Happens when the client closes.
- Event_Show - Happens when it receives an event from *threadShow* (that was sent by the server). Paints and erases from the SDL window based on the message received.
- SDL_MOUSEMOTION - Happens when the player moves the mouse to any place on the game window. If the mouse is on any line orthogonal to the client pacman, it sends a play request to the server (the play is made to the place next to the pacman in the direction of the mouse).
- SDL_KEYDOWN - Happens when any key is pressed. If the pressed key is any keyboard arrow (up, down, left or right) or keys W, A, S, D, it sends a play request to the server.

The function *threadShow* fluxogram is represented in figure 3.5.

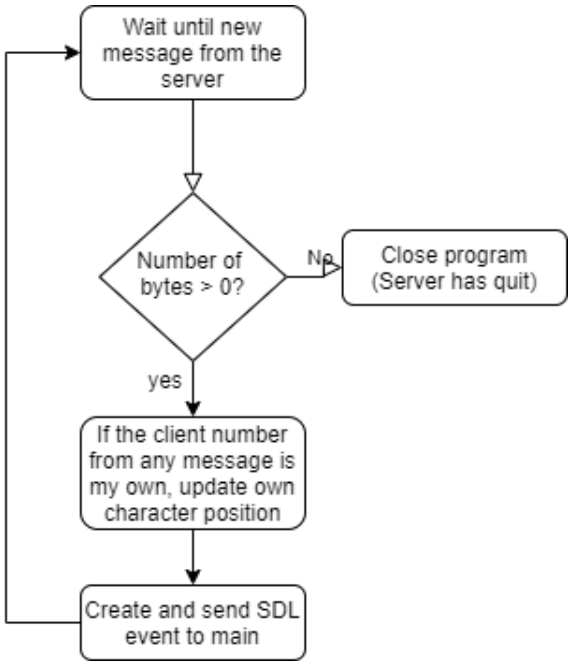


Figure 3.5: *threadShow* fluxogram

4 Communication Protocols

Throughout the program, the server only sends data to the client through structures *messageInit* and $ServMsg$. The client sends an array of 3 integers when informing its colour to the server, and during the game it only sends data/requests over structures *message*.

In both structures - $ServMsg$ and *message* - there is a variable of *char type*. This *char* is used to differentiate the different types of processing to be done. The client can send it as a P, M or S (meaning pacman, monster or super power pacman, respectively), otherwise the message is ignored by the server. In this case, the client informs the server which character to move. However, the server needs to send other characters to clients, namely to identify when a fruit needs to be painted, when a new client entered and when a client leaves the game. This is identified by letters F, N and Q, respectively.

When a client makes a play, it sends a struct *message* with *type* filled accordingly to the played character (keyboard -> monster, mouse -> pacman). When the server receives an invalid play, or a corrupted message, it is ignored and the player doesn't receive any response. When the server receives a valid play and process it, it sends a struct $ServMsg$ to every active client, containing the corresponding character to be painted in *type*, its old and new position, the colour to paint it, and if the client checks that variable $colateralDamage$ is set to 1, means that the character played had effect on other character on the board. This way, variable $colateralType$ is filled the same way as *type*, in respect to the character

it is to be painted, and the information about this character is stored in variables with *colateral* in its name.

The server uses $type = N$ when sending the initial update of the board to a new client, or when sending the characters of a new client to old clients. To inform clients to paint fruits, the server uses $type = F$. In this case, if at any instance it is needed to send two fruit positions at the same time, *colateral* variables are used, and filled with data about the second fruit.

On the server, after any $ClientThreadShow$ thread processes a valid play, it creates and fills a $ServMsg$ structure, and sends a pointer to it to the main SDL queue. Then, the server paints the message content on its own board (running the same code as a client run), and sends the exact same message to every active client. So keep in mind that to send an update to clients, messages always need to go throught the server SDL queue to the main function.

5 Validation

Initializing the main function, an input argument check is made, verifying if the board size is included in the console, as well as if it's made up of integers. Then, it verifies if the stream socket is correctly created, binded and listening for communications. It also ensures that the shared memory is correctly created and attached to the global variable that contains the score board. This is done by checking the returns of the used functions. When creating the board, the server waits for the process that is creating the board (program $board - gen$) to terminate.

Message content verification is done each time the server receives a message from a client, in the function ClientThreadShow. If any information is invalid, the message is to be ignored (server doesn't answer), and the server continues waiting for new messages. Invalid information can be not assigning a valid type of character to a move (P, S or M), or requesting a non-adjacent place (up, down, left or right). This is accomplished by verifying the return value from the function *AnalyseClientPlay*.

As it will be detailed in Section 7 - Synchronization, the synchronization regarding updating the board and the client position is only done in function ApplyGameRules, which is inside that valid message zone. However, if two valid plays happened at the same time with two characters that affect each other, the first client thread that enters the mutex will change the position of the other. When the other enters the mutex zone, there is a verification that the character to move didn't change position since the play was analysed (with function $AnalyseClientPlay$), ignoring the message if it had been victim of colateral damage.

Argument verification was implemented on the client as well, prompting an error message if the input arguments are not valid (<ip> <port> <colour>). This validation is done for each one of the arguments. Also, the client only connects if the server isn't full (if there is available positions to fit one more pacman and monster). Otherwise, a "server is full" message is prompted and the new client is disconnected. This is done in the *threadAccept* function. If the client receives an unrecognizable message from the server (regarding variables *type* and *colateralType*), the message is also ignored.

6 Implemented Functionalities

Control of the pacman and monster on the client

The characters are controlled by each client using SDL events to identify what character and where to move, as explained in *Code Organization* section 3.2. The pacman is controlled by the mouse, and the monster is controlled by the keyboard. After an SDL event on the client (regarding mouse movement or key press), a request is sent to the server, and the client's board is only updated if the server validates and answers to that play. Keep in mind that every client only has it's own position stored in memory.

- Pacman: When receiving an SDL_MOUSEMOTION event, the client checks whether the mouse is up, down, left or right of the current pacman position, and increments/decrements only one pacman coordinate (based on mouse position). Then, it sends a message to the server with this data. To improve the ease of playing the game, the mouse doesn't need to be in any place next to the pacman, instead it can be on any orthogonal line to its position.

- Monster: When receiving an SDL_KEYDOWN event, the client checks what key is pressed. Keys W, A, S, D were mapped to do the same as arrow keys UP, LEFT, DOWN, RIGHT, respectively. The message sent to the server is based on which key is pressed, being the position requested equal to the current monster position, incrementing/decrementing one of the coordinates based on the pressed key.

Character Movement and time between plays

When the server receives a play request from any client, it runs function *AnalyseClientPlay*. This function verifies if the requested play is towards any of the 4 adjacent places of the character to move, and checks if the minimmum time between plays has passed (0.5 seconds). Then, the function validates the play by returning 1. It returns 0 otherwise.

Then, if the play is valid, the server stores the time instance of the played character, and steps into function *ApplyGameRules* (where $SyncVars->muxGameRules$ is locked). The return of this function identifies the type of interaction of the play ('W' -> Wrong character position, 'N' -> No interaction with other objects, 'B' -> Bounce on a brick or end of board, 'C' -> Change positions with other character, 'E' -> Eats/is eaten by a character and 'S' -> SuperPower pacman transformation). This function checks variable $message->type$ of the requested play, and accesses $ClientInfo$ structure to retrieve the current client character position. When this is done, it checks if the position has changed since the play was validated by *AnalyseClientPlay* function (returning 'W' if it has). To analyse a play, a division in 2 phases is done. Initially the server checks if the play goes into a wall or a brick, bouncing it back if so, and only then the other possible interactions will be analysed. The following interactions can happen:

- Empty position - Returns 'N' and simply fills $ServMsg$ with the character to paint, where to erase (old position) and paint (new position) it.
- Against wall/brick - Returns 'B'. There was an initial separation of whether a movement was in x or y axis, and the processing was the same as if it was a brick or a wall. To separate into x or y axis, the condition $y\_new = y\_old$ or $x\_new = x\_old$ had to be verified, respectively. Then, it is analysed if the play is to the left/right or up/down, making the character bounce 2 places in the opposite direction, and it updates on a local variable the new character position. Then, it is verified if the bounced place is inside the board region, keeping the old position if not. If the new place is inside the board, its occupation is analysed, and the other interactions can normally happen (besides, if it goes into a brick again it keeps the old position).
- Change Position - Returns 'C'. Happens when two characters from the same player interact, a pacman or super power pacman interacts with another pacman or super power pacman, or a monster interacts with a monster.
- Monster/Super Power Pacman Eating - Returns 'E'. Happens when a monster and a pacman or a super power pacman and a monster from different characters interact.
- Against Fruit - Returns 'S' if eaten by a pacman or 'E' if eaten by a monster.

Validation of the maximum number of players

The maximum number of players allowed by the server is stored in variable $MAXCLIENTS$, initialized in *ServLib.h*. After the server sets the bricks positions, it stores the number of input bricks, and the

maximmum number of players can be easily obtained applying the following equation:

$$MAXCLIENTS = \frac{(BoardSize\_x \times BoardSize\_y) - numBricks}{4} \tag{6.1}$$

Being $(BoardSize\_x \times BoardSize\_y)$ the number of available places on the board, $numBricks$ the number of input bricks and 4 the number of characters to be inserted when a new player enters (pacman, monster and 2 fruits).

Placement of new players/fruits

When a new client connects, the server creates a new thread to handle communications. Before being ready to receive messages, the server sends an initial message to the client, as explained before. In this initial message, there are variables that indicate the position of its new characters (messageInit->pac[2] and messageInit->moob[2]). The server generates the new positions using function $generateValidRandomPos$.

This function is defined in $ServLib.c$, and runs a cycle that generates random numbers between 0 and the size of the board in x and y. Then, inside a zone locked (to read) by $SyncVars->rwLockBoard$, it checks if the generated position is currently empty. If so, the function returns. If not, it repeats the cycle. This function is used throughout the code whenever there is the need to generate a new unoccupied position (either when a character is eaten, when a new client enters, when a new fruit needs to be spawned or even when the inactivity reset is triggered).

Client disconnect

Guarantee of 2 movements per second max.

This functionality was implemented both on the server and on the client, to prevent the server from receiving excessive messages. It uses function $gettimeofday$ from library $time.h$ to store the time instance of a play. On the server, if a thread receives a valid play request, it stores its time instance on the respective $ClientInfo$ structure. Then, when receiving a new play, it verifies if 0.5 seconds have passed since the last valid play of the requested character.

On the client, if a message containing its client number is received (not including colateral variables), it stores the time instance, and only allows a new request to be sent to the server (from the respective character) after 0.5 seconds.

Character inactivity

The character inactivity functionality was implemented in $ClienThreadShow$ by using function $setsockopt$ to set a maximmum time of 30 seconds for the socket to block on receive. When a thread handling a client

is blocked for 30 seconds since the last received request, the socket receive returns -1 and the server generates new random position for both player characters.

Fruit Management

Thread $ShowFruitsOnBoard$ manages the number of fruits on the board. When there is nothing to be done regarding fruits, the thread waits to be signaled on a condition variable without wasting processor cycles. The signal is sent everytime a client connects/disconnects or a fruit is eaten by a character.

When a signal is received, the server verifies if the number of fruits on the board is correct based on the number of active players ($numFruits = 2 * (numActiveClients - 1)$). If new fruits need to be painted, the server generates 2 new valid random positions to paint them, and if fruits need to be erased, the server clears the ones placed on higher and lower x and y coordinates (the "first" and "last" fruit on board). Then, the thread updates the board structure, stores a local variable with the number of fruits currently on board, and sends an event to the SDL queue containing data about the fruits to be painted.

If the number of fruits on board is correct, then the server identifies the received condition variable signal as an event of an eaten fruit. In this case, a timer of 2 seconds is created to paint the missing fruits on the board. Function $paintFruitEvent$ creates the propper $ServMsg$ to be sent to identify fruit painting.

Then the thread is blocked on the condition variable until a new signal is received.

Superpowered pacman

The super power mode is identified on variable $ClientInfo\text{->}superPower$, and the number of possible eats before returning to normal mode is stored in $ClientInfo\text{->}superPowerEats$. When a pacman eats a fruit, it activates super power mode, and the server sets variable $superPower$ to 1 and variable $superPowerEats$ to 2. For every monster eaten by a super power pacman, the server decreases the number of $superPowerEats$. If this variable reaches 0, the client's pacman goes back to normal, setting variable $superPower$ to 0. If another fruit is eaten while still in super power mode, the variable $superPowerEats$ is reset to 2.

When a player moves a pacman, it sends to the server a message with variable $type = P$. However, when the server processes that message, it will check if the super power mode of that client is active, and updates the *type* of character to 'S' to allow the super power interactions to happen.

Game score board

The game score board is stored as a shared memory, to allow multiple programs to access it. The server writes to the shared memory every time a pacman is eaten by a monster, a monster is eaten by a

super power pacman or a fruit is eaten by any character.

The clients display the score board on the terminal every minute. To synchronize all clients, the server stores the time instance of the creation of the shared memory on variable $ScoreBoardInit$, and every time a new client connects, the server calculates the remainder of the division of $ScoreBoardInit$ by 60 seconds. The remainder of this division identifies how many seconds are left before the next score board display, and the server sends the remaining time in variable $messageInit$ ->$timeToShowScore$ (this way, every client is synchronized with the "server time"). The client uses this value to generate a timer to trigger SIGALRM and after the first score board display, the client sets a new alarm of 60 seconds.

Data cleanup

Data about the connected clients is stored in an array of structures $ClientInfo$, with $MAXCLIENTS$ elements, allocated using a *malloc*. This way, when a client disconnects, the server sets variable *activeClient* to 0 on the respective structure, and when a new client connects it can be assigned to any structure with $activeClient = 0$. Thus, the memory addresses used for a client that has disconnected are reused by a new connected client.

Threads that handle client communications are deleted whenever the respective client has quit, and new ones are created when a new client connects.

Every $ServMsg$ is created using a *malloc*, and when the server sends the data in it to the connected clients, it frees the correspondent memory region.

When the server closes, it destroys all initialized synchronization variables, deletes the sockets file descriptors, frees $ClientInfo$ array of structures and removes the shared memory segment, unmaping the score board structure from it.

7 Synchronization

As explained in *section* 2, all used synchronization variables are stored in structure $SyncVars$. R/W lock $rwLockBoard$ is the variable that guarantees that no thread is writing to the board structure while other threads are reading from it, and that no thread reads while others are writing. The server locks this variable on write whenever there is a character position update, in which it has to clear one place and/or fill another (used in functions $generateInitialMessage$, $ApplyGameRules$, $paintFruitEvent$, $ClientThreadShow$ and $ShowFruitsOnBoard$). Read locks are used whenever there's a need to check if a place is occupied and by which character (used in function $generateValidRandomPos$, $ApplyGameRules$

and $ShowFruitsOnBoard$).

R/W lock $rwLockClientInfo$ is used to guarantee that a client's character position (either variable x_pac, y_pac, x_moob or y_moob) is only read when no thread is writing to it. This variable is locked on write when there's a character position to update, by functions $GenerateInitialMessage$, $ApplyGameRules$ and $ClientThreadShow$. The server locks this variable on read when there's a need to know the position of a character from any player, in functions $ApplyGameRules$, $UpdateOldClientsScreen$, $UpdateNewClientScreen$ and $ClientThreadShow$.

To prevent errors related to the assignment of a new place in the $ClientInfo$ array, variable $rwLockActiveClient$ is used. The server locks it on read to verify the state of variable $ClientInfo$ ->$clientActive$ when assigning a new client to an index. When a client enters, the server locks the variable on write to set $clientActive$ to 1, and when a client quits, it also locks the variable on write to set $clientActive$ to 0.

In order to avoid errors when receiving multiple requests at the same time, $muxGameRules$ is used to guarantee that only 1 thread is analysing the interaction of a play. It is only used in function $ApplyGameRules$, and covers the whole code that process the effects of a play on the board, being only unlocked after the update of structures $board$ and $ClientInfo$.

Mutex $muxFruits$ is mainly used to allow the conditional variable $condFruits$ to be put on hold when there's no work to be done, and to be signaled when a client connects/disconnects or a fruit has been eaten.