Systems Programming

2022/2023

Project Assignment – Part B

BetterChaseThem

In the second part of the project students will complement the previous project, by solving some of the identified issues with communication, rules implementation and usability. This new version of the game is called the **BetterChaseThem**.

In this part of the project, students will use threads and **Internet domain stream** sockets.

The rules of the game will be the same with minimal changes on the dead of a ball.

All the assumptions and rules described in Part A of the project still apply except if explicitly contradicted or voided; the changes are identified as a line in the margin.

1 BetterChaseThem Rules

In the **BetterChaseThem** game, users control a ball (represented by a letter) that moves in a field. In those fields, bots (represented by *****) move randomly. Prizes (numbered from 1 to 5) are available to be eaten.

Each ball has a health value that varies as follows:

- when a ball rams into another one, there is an exchange of health:
  - the ball that moved/rammed onto another one increases its health by one point;
  - the other ball (the one that was rammed) decreases its health by one point;
- when a ball rams into a bot, nothing happens;
- when a bot rams into a ball, the ball's health decreases by one value;
- when a ball eats a prize, its health increases by the value of the prize.

A ball health starts at 10 and is never higher than 10 (even if eating a lot of prizes). After the health reaches 0 the user has 10 seconds to confirm that he wants to continue to play (just like arcade games).

If the user confirms that he wants to continue playing, the ball health becomes 10, and it can start to move and interact with other balls, bots and prizes. If the user does not confirm, the ball and all its information is deleted, and the client disconnects.

During the 10 seconds wait for confirmation (or until the user confirms) the ball is immobile and any interaction with it is prohibited (balls and bots bumping into are not affected).

2 Project Part B

In the second part of the project, students will implement the **BetterChaseThem** game using a distributed client-server architecture, where each ball is controlled by a user. An independent process/program reads the keys pressed by the user, sends to the server the ball movements, receives the field changes, and updates the screen.

Several users can play simultaneously, and multiple bots should also be moving randomly in the field.

A new prize is placed on the board every 5 seconds.

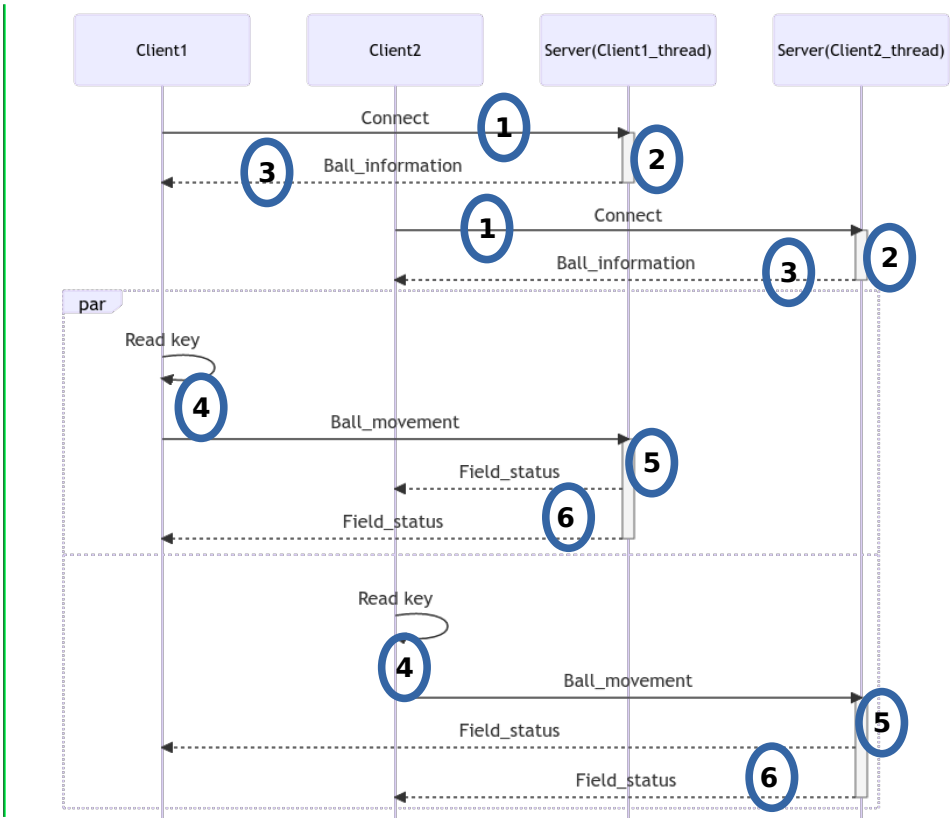
The balls clients should be independent processes/clients that interact with the server using **Internet domain stream sockets**.

The way to move bots and place prizes should be implemented inside the server, not resorting to external processes.

Every ball should be identified by a unique letter. This letter can be assigned by the server or selected by the user left as a programmer decision.

2.1 Interactions

The essential interactions between the ball clients and the server are presented in the following figure.



The server is waiting for socket connections from the clients. After a client is connected a new thread (client_thread) is created and assigned to the control of the client ball.

Each client_thread is waiting for messages from the assigned client, depending on the received message the game changes its state, replies to the client accordingly, and updates all connected clients.

After the socket connect, the basic messages exchanged between the ball clients and each client_thread are:

- **Ball_connect** (from the client to the server)
- **Ball_information** (from the server to the client)
- **Ball_movement** (from the client to the server)
- **Field_status** (from the server to all the clients)
- **Health_0** (from the server to the client)
- **Continue_game** (from the client to the server)
- ~~**Disconnect** (from the client to server)~~ This message is no longer needed and should be replaced by the socket close).

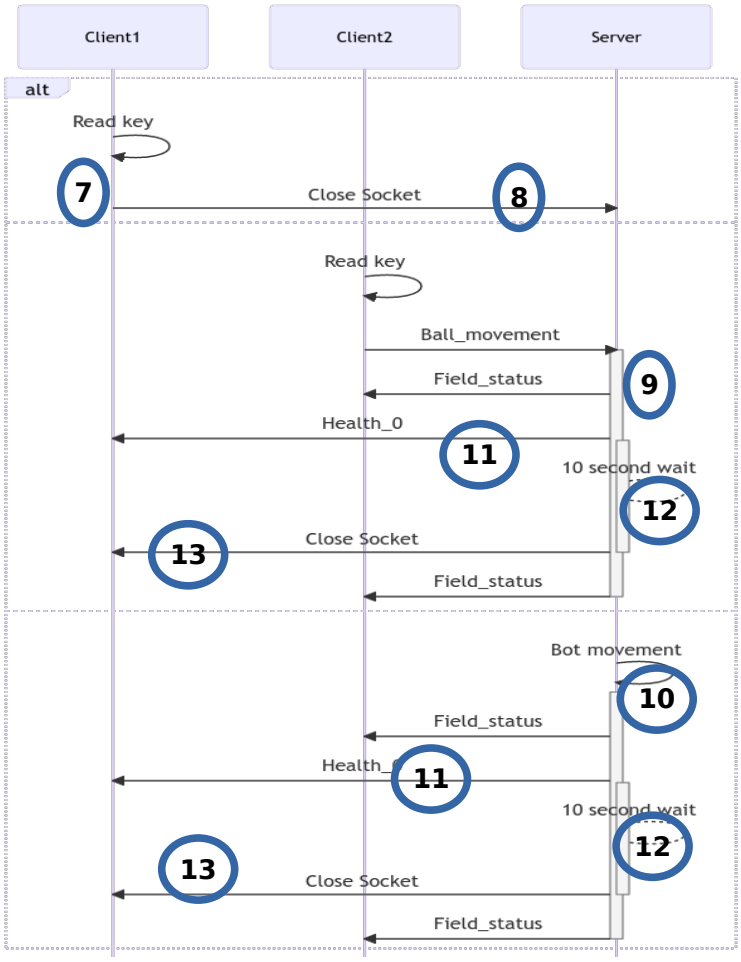
After the server has accepted the socket connection, the ball client needs to send a **Ball_connect** message (**step 1**) at startup.

The server stores the relevant client and ball information in a list (**step 2**) and replies with a **Ball_information** message (**step 3**). After receiving the **Ball_information** message from the server, the user can now start controlling its player.

Every time the user moves its ball on the client (**step 4**), a **Ball_movement** message is sent to the server. The server updates the field where various balls and bots may be playing (**step 5**) and sends the new status of the field as a **Field_status** message to all the clients (**step 6**).

~~If two users/clients are connected, the server will only process one request (**step 5**) at a time and only send the update of the field to the player whose movement is being processed. After processing a ball movement, the server will read a new message and process it.~~

Even if a user/client does not send any **Ball_movement** message to the server, its screen will be updated any time other ball, bots or prizes are updated/move (**step 6**).



There are two ways for a player to be removed from the game:

- **by initiative of the user** – in this case, after reading **Q** from the keyboard (**step 7**), the client will close the socket (**step 8**) and then exit. On the server, the thread assigned to that ball should also terminate orderly.
- **when its health reaches zero** – The health of a ball can decrease and can reach 0 when another ball (**step 9**) or bot (**step 10)** rams into it. If a ball reached health 0, the server should send to the corresponding client the **Health_0** message (step 11). If, in the following 10seconds (**step 12**), the client does not

> send a **Continue_game message** the server should close the socket assigned to such ball and terminate the corresponding thread (**step 13**). The client should also terminate after the socket is closed.

The previous diagram does not represent the clients' threads, because there are multiple ways (in different threads) to implement the presented functionalities. When a client/ball is removed from the game (**Close Socket** or **Health_0**), the server should update accordingly its information in the array/list that stores all client information and ignore any other message from such client.

Students can slightly modify the presented interactions, or implement other message exchanges, if necessary to implement the described functionalities.

2.2 BetterChaseThem Server

The **BetterChaseThem server** is a C program with a **Internet domain stream socket** to receive connections from various clients wanting to play the **BetterChaseThem** game.

The maximum number of simultaneous players is limited by the space availability in the game field. Students should decide what happens to a client that tries to connect but there is no available space in the game.

The server should store a list/array of all the clients with all the relevant information (e.g., player position, health, ...). A client is inserted into this list when the client connects and is removed when client socket is closed.

When the server receives a **Ball_movement** message, a bot moves, or a ball is removed, it should calculate and update the whole global state of the field (prizes and other players health) and inform all clients about the new field status.

The maximum number of simultaneous bots is ten (10).

To simplify the test of the project, the address and port of the server socket should be supplied to the server program as a command line argument.

2.3 BetterChaseThem ball Client

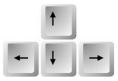
The **BetterChaseThem ball client** is a C program that interacts with a server using **Internet domain stream sockets**. This program allows a user to control a ball on the game field.

The address and port of the server should be supplied to the program as a command line argument.

This program reads key presses from the keyboard and forwards them to the server (if corresponding to the cursor keys) to move the user ball.

After Connecting to the server and being accepted to the game, the client will wait for one **Ball_information** message. Only after receiving this message the client should start concurrently to:

- read a key presses, and send **Ball_movement** messages to the server;
- receive messages from the server and update the client display with the new state of the Field.

The ball is controlled using the cursor keys: 

If the user presses the **Q** key, the client should terminate and close the socket.

The client should terminate (exit) after the user presses the **Q** key or when the socket is closed by the server.

All clients should update their screen synchronously whenever there is a field status change: new ball, ball movement, bot movement, prizes changes, and ball removal.

2.4 Bot update

The decision to move bots should be performed inside the server, not requiring a separate client.

Every 3 seconds, the server decides in what direction the bots will move, updates the bots positions and the health of any affected balls. All clients should be notified of the new field status (message **Field_status**) and update their screens accordingly.

2.5 Prizes

During the game, there should exist between zero and ten prizes; each prize has a value between 1 and 5.

When the game starts, 5 prizes are placed randomly in the field.

Every time a ball moves into a prize, the ball's health is increased by the value of the prize, and the prize disappears.

Every 5 seconds a new prize should be put into the field, up to a maximum of 10..

Students should decide how to place a new prize every 5 seconds and how to guarantee that there are at most ten prizes in the field.

The decision to place prizes should be performed inside the server, not requiring a separate client.

2.6 Ball/bot movements

The ball and bot positions should be calculated on the server after receiving messages from the **BetterChaseThem ball clients** or any bot movement.

Balls and bots should:

- not share the same space (when moving or being randomly placed);
- not bounce if the ball/bot goes against a wall (it should count as a movement, but stay at the same place);

- remain in the same place when a ball/bot rams into another ball/bot.

When a ball eats a prize, the prize will disappear, and the ball should go into its place. Bots and prizes do not interact; when a bot rams into a prize it constitutes a movement, but it does not eat it and should both remain in the same place.

2.7 User interfaces

The **BetterChaseThem ball client** will use **ncurses** to read the keyboard and draw the balls, bots and prizes on the screen.

All clients should update their screens synchronously whenever there is a change on the server game field (after receiving a **Field_status** message).

The **BetterChaseThem server** should also display the game status and update the screen whenever a field update is done.

The client interface should follow the same structure as the one of part A.

As the server receives messages it should update the location of the various balls, prizes, bots and players' health, by deleting the characters on the old positions and writing the corresponding symbols in the new positions.

The size of the field and message can be defined as constants in a **.h** file to be included by the clients and server (as in the provided example code).

3 Project Development technologies

For the communication of the clients with the server, students should only use **Internet domain stream** sockets.

Students should implement the system using standard C (ANSI or C99) **without** resorting to the following:

- ~~threads;~~ Students can now use threads!
- select;

- non-blocking communication;
- active wait.

Students are expected to use **pthreads** and pthreads synchronization primitives.

4 Race conditions

On the server, the field representation, the ball, bots and prizes, can be updated concurrently by various threads. This generates race conditions when multiple threads update the same data.

To guarantee that any race condition is solved and that the game works as expected, students should implement suitable synchronization.

5 Error treatment / Cheating

When implementing a distributed/network-based system, servers cannot guarantee that the clients lawfully abide to the defined protocol.

If communication protocol allows it, malicious clients can use the developed messages and interactions to cheat.

Besides the verification of all the received messages on the server to detect errors in communication, the protocol and data exchanged between clients and server should guarantee that no cheating can be performed by a malicious client that subverts the semantic and order of the messages.

6 Project submission

The deadline for the submission for part B of the project will be **13th January 2023 at 19h00 on FENIX**.

Before submission, students should create the project group and register at FENIX.

Students should submit a **zip** file containing the code for both games. Since the complete system composed is composed of a server and multiple clients, each of the

developed programs should be placed in different directories. One or multiple Makefiles for the compilation of the various programs should also be provided by the students

7 Project report

Along with the project code, students should also submit a report detailing all the functionalities implemented in Part A and part B of the project.

A report template and further instructions will be provided to the students.

8 Project evaluation

The grade for this project will be given taking into consideration the following:

- Number of functionalities implemented
- Communication
- Synchronization
- Code structure and organization
- Error validation and treatment
- Cheating robustness
- Comments