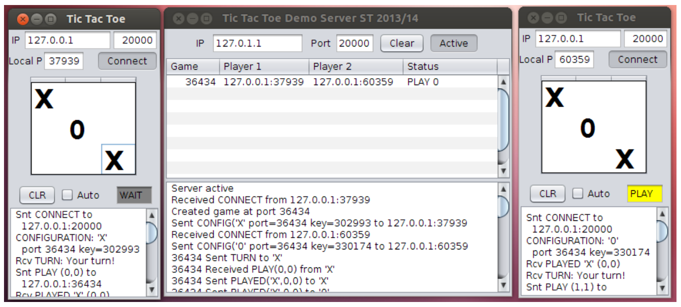
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Electrical Engineering Department



| Game | Player 1 | Player 2 | Status |
|---|---|---|---|
| 36434 | 127.0.0.1:37939 | 127.0.0.1:60359 | PLAY 0 |

```
Snt CONNECT to
  127.0.0.1:20000
CONFIGURATION: 'X'
  port 36434 key=302993
Rcv TURN: Your turn!
Snt PLAY (0,0) to
  127.0.0.1:36434
```

```
Server active
Received CONNECT from 127.0.0.1:37939
Created game at port 36434
Sent CONFIG('X' port=36434 key=302993 to 127.0.0.1:37939
Received CONNECT from 127.0.0.1:60359
Sent CONFIG('0' port=36434 key=330174 to 127.0.0.1:60359
36434 Sent TURN to 'X'
36434 Received PLAY(0,0) from 'X'
36434 Sent PLAYED('X',0,0) to 'X'
```

```
Snt CONNECT to
  127.0.0.1:20000
CONFIGURATION: '0'
  port 36434 key=330174
Rcv PLAYED 'X' (0,0)
Rcv TURN: Your turn!
Snt PLAY (1,1) to
```

# Telecommunication Systems

## 2013/2014

**Laboratory Work 1:**
**Application using sockets – the tic-tac-toe game**

***Integrated Master in Electrical Engineering and Computers***

**http://tele1.dee.fct.unl.pt**

# Index

## 1. OBJECTIVE

**Familiarization with the use of sockets for communication between machines and the use of the programming interface of Java sockets.**

The work consists in developing a distributed tic-tac-toe game, where clients communicate with the server using datagram sockets.

> **Suggestions**: In certain parts of this document appears some text formatted differently that begins with the word "Suggestion". It is not mandatory to follow what is written there, but may be important for students or groups where there is not yet at ease with programming, data structures and algorithms.

## 2. SPECIFICATIONS

You will implement a distributed tic-tac-toe game. The game server behaves as a portal for clients because they have to know only one address and access port. Every game that is in progress has a port number not publicly known. Communication between the clients (players) and servers is performed exchanging datagrams.

The general idea is as follows: the clients access to the access server (*portal*) for the address and port of the game (*game*) server, and a key (*key*) that will have to provide when accessing the game server. The access server will dynamically launch new game servers, matching client pairs. The first player plays as 'X' and the second as '0'. Each game server will manage all interactions between the two players, indicating when it is time to play ("TURN"), receiving and validating the plays ('PLAY' message) from the players and sending messages informing about the plays ('PLAYED'). When the server detects the end of the game (when a user makes three in a line, or all the board positions are occupied) sends the message ('ENDED'), ending the game.

The client application represents graphically the state of the game, receives user commands and executes all iterations with the access server and the game server to implement the game.

### 2.1. MESSAGE FLOW

The message sequence during a game is shown in Figure 1, and consists of the following messages (assuming that no errors exist of the type of invalid service, invalid keys, etc.):

(1a e 1b) – One player sends a connect request message ("CONNECT") to the access server's socket;

(2a e 2b) – The access server processes the connection request and (2a) launches a new server for the first player, who plays as 'X', or (2b) associates the player to an existing game, playing as '0'. In this second case, the game server starts with the algorithm of the game;

(3) – The game server sends a message "TURN" to player 'X', signaling that it is the next one to play;

(4) – After the player selects the next move, it sends the message "PLAY" with the selected position "(0.0)" and waits;

(5a e 5b) – The game server validates the move from player 'X' and sends a message "PLAYED" for the two players, indicating the last position played. Then …

(6) – …the game goes to the player '0 'with a TURN message;

(7) – After player '0' chooses the position, he generates a PLAY message, which will be echoed to the two players (8a) and (8b) if it is valid;

Sequence (3) - (8) it will be repeated until one of the players make 3 in a row, or all positions of the board are occupied. Assuming that player 'X' makes the winning move in (9), ...

(10a and 10b) – … the game server signals the end of the game and the result through an END message, which could have "Won", "Lose" or "Tie" values in case of a tie.

The communication with the game server uses datagram sockets. In this work it will be assumed that packets are not lost and that no errors occur in the communication, although it may happen in this type of socket.

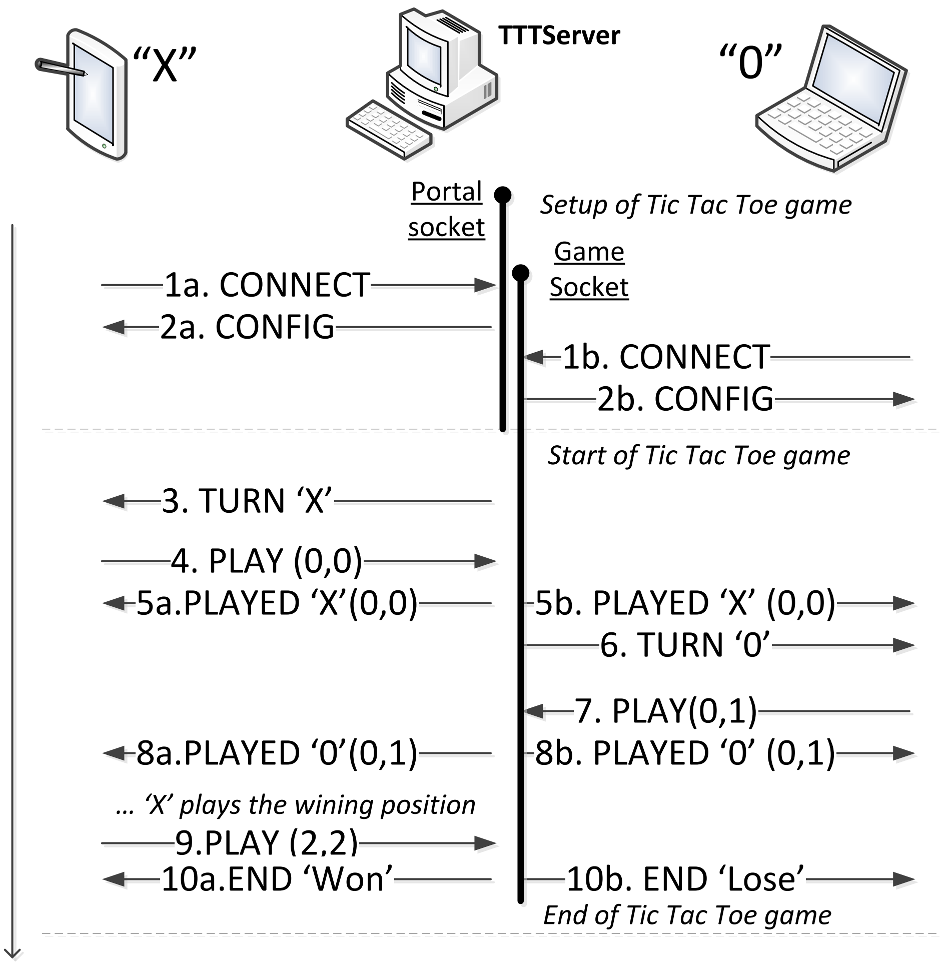
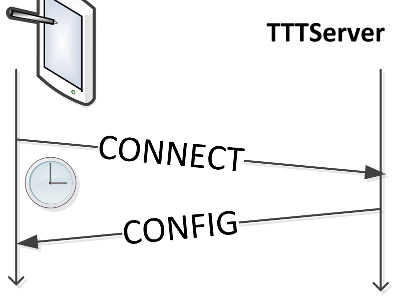


**Fig 1 – Message sequence during a game**

## 2.2. Access server (portal)

Access to the access service includes messages 1a-2a and 1b-2b of Figure 1, which obey to the following request-response protocol:



```
Message CONNECT:// sequence of
 short type;  // Type of message = 99
   // = TTTConfig.PCKT_CONNECT

Message CONFIG: // sequence of
 short type;  // Type of message = 98
   // = TTTConfig.PCKT_CONFIG
 int key; // shared game key
 char player; // player 'X' ou '0'
 int gamePort; // game server port
```

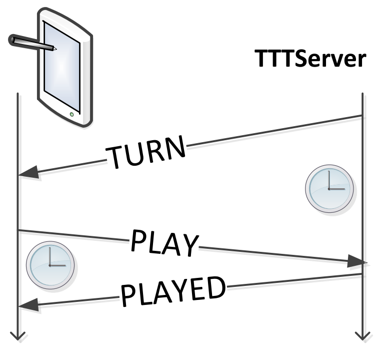
The **CONNECT** and **CONFIG** messages have the parameters presented. The **CONFIG** message contains the random key *key* defined by the game server, how the player plays (starts the game for *player* = 'X', or is the second player to play with *player* = '0') and *gamePort* contains the game server's port. The client uses a timer (*timer*) to detect when the server response is delayed more than 10 seconds, ending the game.

As mentioned, the access server socket is the only one known. Usually, it is called this a ***well-known socket*** (the IP address is that of the respective machine and what is really known is the port). All other sockets associated with games must have free ports since their values are provided to the various participants in the games. The port of the first running access server socket is **20000**. If more than one server is run on the same machine, they will have the numbers 20001, 20002, etc..

**Sugestion 1 – Several active games**
Initially you may assume that a server can support only one game at a time (if it seems that simplifies programming). If you have time, you should prepare the server to support multiple concurrent games. Using a list to store the active games is recommended, since it allows ending all active games. Do not forget that in any circumstances the main thread (linked to the graphical interface) should not be blocked. Any blocking operation (e.g. reading a socket) must be held within a separate thread.

## 2.3. GAME – PLAYING

The access to the game service is illustrated by messages 3-5 and 6-8 of Figure 1, which follow the protocol:



```
Message TURN: //sequence of
 short type;  // Type of message= 10
   // = TTTConfig.PCKT_TURN
 int key; // shared game key
 char player; // player 'X' or '0'

Message PLAY: //sequence of
 short type;  // Type of message= 20
   // = TTTConfig.PCKT_PLAY
 int key; // shared game key
 char player; // player 'X' or '0'
 byte row; // line = 0, 1 or 2
 byte col; // column = 0, 1 or 2

Message PLAYED: // sequence of
 short type;  // Type of message= 11
   // = TTTConfig.PCKT_PLAYED
 int key; // shared game key
 char player; // player 'X' or '0'
 byte row; // line = 0, 1 or 2
 byte col; // column = 0, 1 or 2
```

The game server sends the TURN message to the next player to play and waits a maximum time of 20 seconds for his move. The player sends the move through a PLAY message that if valid, is echoed to the two players through a message PLAYED. All messages contain the key shared with the user (*key*) and the identification of player playing (*player*). The PLAY and PLAYED messages contain the coordinates of the position of the move (*row*, *col*).

**Suggestion 2 – play validation**: The game server may validate that the IP and port correspond to the socket of the player who sent CONNECT and if he plays in its turn.

**Suggestion 3 – number of plays out of turn**: The client application may allow players to play out of turn. The game server must detect this event, and if there are more than 4 plays out of turn, the client must be punished with the loss of the game.

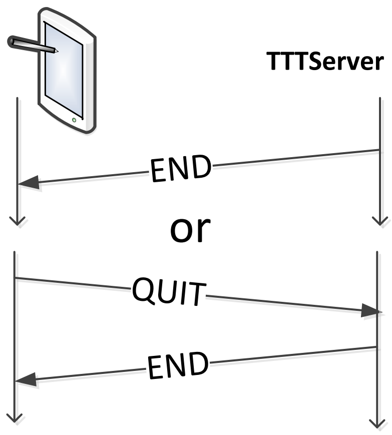
**Suggestion 4 – maximum waiting time**: If there is time, the game server should validate that the maximum time to send the PLAY message is respected. In the first phase of implementation, it is recommended that this validation is not done.

### 2.3. GAME – END OF THE GAME

The game ends when a player gets three positions in a line or if all positions are marked, and the server generates an END message, shown in message 10 of Figure 1, with the structure shown below. The END message does not include the player's name but includes the result code *res* indicating the outcome: if the player won, lost or tied the game.

At any time, a player can quit the game (loosing the game) by sending a QUIT message, which causes the end of the game.

Em qualquer altura, um jogador pode desistir do jogo (ficando derrotado) enviando uma mensagem QUIT, que provoca o fim do jogo.



```
Message END: //sequence of
 short type;  // Type of message= 90
   // = TTTConfig.PCKT_END
 int key; // shared game key
 byte res; // resultado para o jogador:
   // TTTConfig.END_GAME_WON= 1  - Won
   // TTTConfig.END_GAME_LOSE= 2 - Lose
   // TTTConfig.END_GAME_TIE= 3  - Tie

Message QUIT: //sequence of
 short type;  // Type of message= 97
   // = TTTConfig.PCKT_QUIT
 int key; // shared game key
 char player; // player 'X' or '0'
```

## 3. PROGRAM DEVELOPMENT

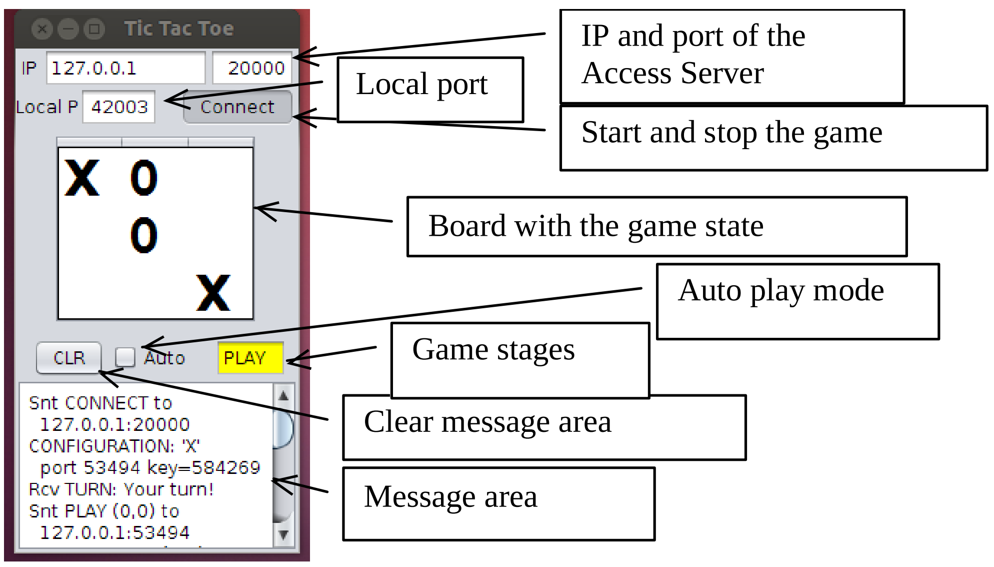
### 3.1. CLIENT

The client application is provided entirely implemented together with the statement of work. A Java application was developed with the graphical interface shown below.

When you press the "Connect" button, the client creates a datagram socket with the port indicated in *Local IP*, and communicates with the access server at the IP address and port shown in the first line of the GUI, performing all steps shown in Figure 1.

The GUI of the client application shows a board with the game state and indicates which is the stage of the game (i.e. whether it's time to play or to wait). The *Auto* button selects whether the client plays randomly or if the user selects the positions to play.

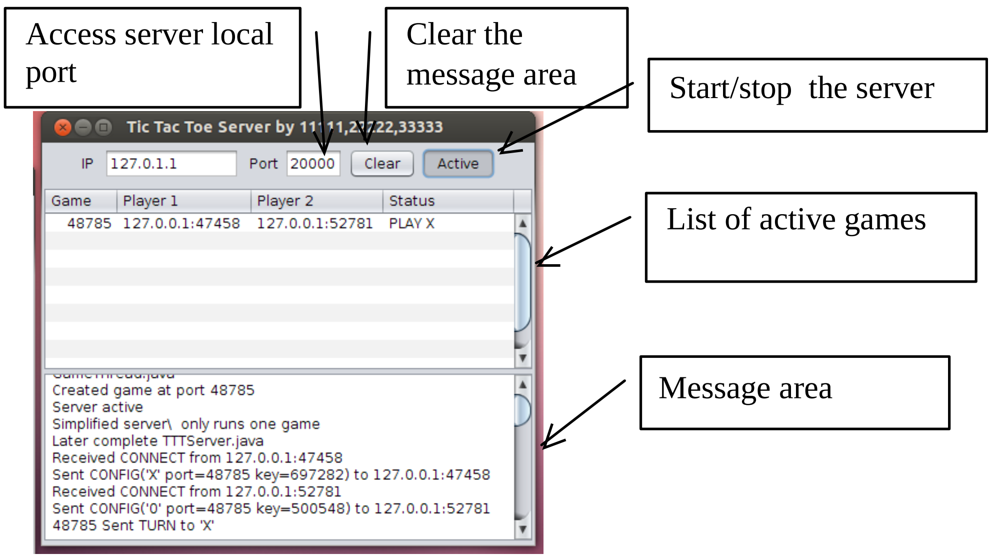
You can run the client from the terminal using the command: `java –jar TTTClient.jar`



```
Snt CONNECT to
  127.0.0.1:20000
CONFIGURATION: 'X'
  port 53494 key=584269
Rcv TURN: Your turn!
Snt PLAY (0,0) to
  127.0.0.1:53494
```

## 3.2. Server

The work consists of implementing the server block. To facilitate the development of the program and make possible the development of the program in three classes, it is provided an incomplete TTTServStudents program with the graphical user interface shown below, which already performs part of the functionality requested. Each group can make all the changes you want to the basic program, or even draw a new GUI. However, it is recommended that you invest the time in the correct implementation of distributed application proposed.



| Game | Player 1 | Player 2 | Status |
|---|---|---|---|
| 48785 | 127.0.0.1:47458 | 127.0.0.1:52781 | PLAY X |

```
Created game at port 48785
Server active
Simplified server\ only runs one game
Later complete TTTServer.java
Received CONNECT from 127.0.0.1:47458
Sent CONFIG('X' port=48785 key=697282) to 127.0.0.1:47458
Received CONNECT from 127.0.0.1:52781
Sent CONFIG('0' port=48785 key=500548) to 127.0.0.1:52781
48785 Sent TURN to 'X'
```

Five classes, distributed by two packages, compose the server program provided. Package *game* is used by both the client and the server and has the classes:

- *TTTConfig.java* (complete) – Class with the configuration of the application, which defines constants and static functions;
- *GameState.java* (complete) – Class that memorizes the game state, detecting wining conditions and the end of the game.

Package *server* implements the server specific functionalities. It has the classes:

- *ConnectThread.java* (complete) – Thread that receives packets at the access server's socket;
- *GameThread.java* (**to be completed**) – Thread that implements the game server and receives packets in the game server's socket, storing all the game's data;
- *TTTServer.java* (**to be completed**) – Main class with the GUI, which manages all the objects used.

The supplied server program already performs the first three interactions (1.-3.) of Figure 1. It is intended that students complete the code in order to implement all functionalities described in section 2 of this document. The code provides a set of support classes and methods, which are described below.

### 3.2.1. Support classes and methods

Class *TTTConfig*

Class *TTTConfig* of package *game* is used to aggregate all the constants and associated functions used in the protocols. It defines the types of messages and the types of results sent in the END message, previously introduced in section 2. Moreover, it defines two helper methods that allow you to write legibly the value of a type or a result:

```
/** Returns a string with the name of the packet type */
public static String type_str(short type);

/** Returns a string with the result of a game */
public static String end_result_str(int result);
```

Class *ConnectThread*

Class *ConnectThread* implements the reception of datagram packets at the access service. This class is very similar to the *Daemon_udp* class shown in the second class of laboratory work 0. The main difference is that the message type is decoded and the method *handle_CONNECT_packet* of class *TTTServer* is called only when it receives the *TTTConfig.PCKT_CONNECT* type.

Class *GameState*

The *GameState* class stores the state of a game, and allows to detect if the game is over, and determine who is the winner, or if there is a tie. With the *play* method, you can validate whether each individual move is valid. Thus, it will be used an object of this class for each active game.

```
public class GameState {
  private final char[][] state;  // Game board state

  public void State();  // Constructor

  /** Regist a play from one player in the game state
   * @param player  name of the player: 'X' or '0'.
   * @param row   row number: 0, 1 or 2.
   * @param col   column number: 0, 1 or 2.
   * @return true if it is a valid play, false otherwise.
   */
  public boolean play(char player, int row, int col);

  /** Return the player who played in a position of the board
   * @param row   row number: 0, 1 or 2.
   * @param col   column number: 0, 1 or 2.
   * @return 'X', '0' or ' ' if no one played in the position
   */
  public char get(int row, int col);
```

```
    /** Test the board to detect any invalid position
     * @return true if valid, false otherwise.
     */
    public boolean check_valid_game();

    /** Detect if there is a winner of the game
     * @return the winner name ('X' or '0'), or ' ' if it is a tie or it hasn't ended.
     */
    public char check_winner();

    /** Test if the game has ended
     * @return true if it ended, false otherwise.
     */
    public boolean check_game_ended();
  }
```

Class *TTTServer*

The *TTTServer* class is almost complete. The interaction with the GUI is accomplished through the following set of functions. It includes a function to echo messages to the text box and others that control the table of active games:

```
  // Writes the message in the GUI and in the command line
  public synchronized void Log(String s);

  public int GUI_find_game(int port);  // Return the line with 'port' in the game's table

  // Adds a new game to the game's table and the player 'X' description
  public boolean GUI_add_new_game(int port, String playerX, String status);

  // Adds the player '0' description to a game in the game's table
  public boolean GUI_add_player_0(int port, String player0);

  // Update the state of a game in the game's table
  public boolean GUI_set_state(int port, String status);

  public boolean GUI_del_game(int port); // Delete a game from the game's table

  public void del_all_games(); // Delete all games from the table
```

It defines the *send_CONFIG* method that sends a CONFIG message, which uses the IP address and port that are already configured in *dp* parameter to set the destination of the packet.

```
  private boolean send_CONFIG(DatagramPacket dp, int port, char player, int key);
```

This class also defines some constants, as the maximum time waiting for a PLAY message after sending TURN (*TTTServer.PLAY_TIMEOUT* in ms) and the maximum number of plays out of order tolerated (*TTTServer.MAX_FAULTS*). Most of the remaining functions will require minor modifications for those who perform all exercises.

Class *GameThread*

The *GameThread* class implements the game server and keeps all information concerning the state of a game. It was based on class *Daemon_udp*, but it creates the socket *ds* locally and handles locally the incoming packets.

An important part of this class is the management of information about the players. For this purpose, it defines the inner class *PlayerInfo* and an array of two objects where it stores the information about the 'X' player (at position 0) and about the '0 'player (at position 1). Furthermore, it defines the following set of functions to get and modify the information:

```
  class PlayerInfo {
          InetAddress ip;
          int port;
          int key;

          PlayerInfo(InetAddress ip, int port, int key);
  };
```

```
private PlayerInfo[] players; // Array with the player’s information
private int player_count; // Counter with the number of players registered

public char player_name(int pl); // Return the char associated to the player number
public int player_number(char player); // Return the number associated to the player
public int get_count(); // Return the number of players registered
public int get_port();  // Return the port number of the socket of this game
public int get_key(int pl); // Return the key associated to a player

// Add a player to the game; returns the number of players in the game
public int add_player(InetAddress _ip, int _port);
```

This class also provides methods fully implemented that send the messages TURN and END to one of the players, but the reception of the messages and the sending of message PLAYED are incomplete.

```
// Prepare and send the TURN message to player to
private boolean send_TURN(int to)

// Prepare and send END message to player to
private boolean send_END(int to, byte result)
```

The class already supports methods to start and finish the game, and there is the *active* object field that indicates whether the game is active. Note that a game is not active when only one player is connected, but this thread must allow that player to give up the game sending QUIT. The method *start_game* sends the message 3 shown in Figure 1. The method *stop_game* sends the messages 10a and 10b shown in Figure 1.

```
private boolean active;  // The game is active

public boolean start_game();  // Start the game

public boolean stop_game(char winner);  // Stop the game; for TIE put winner=’ ‘
```

### 3.2.2. Tasks

The work is organised in a sequence of tasks:

**TASK 1) Receive message PLAY**

The *run* method of class *GameThread* already partially decodes the received packets, but it does not completely decode the PLAY message. The first task is to complete this function in order to read the fields *row* and *col*, and then invoke the function that handles the PLAY message.

**TASK 2) Send message PLAYED**

You must implement the *send_PLAYED* method of class *GameThread*, in order to prepare and send a message PLAYED.

```
private boolean send_PLAYED(int to, int p, int row, int col);
```

**TASK 3) Implement the game’s logic**

This task is divided into two steps: First you need to define and initialize fields (variables) in class *GameThread* that save the game state and memorize who's next to play; a second step is to program the *handle*_PLAY method in the same class, so as to perform the game logic. You must validate if the player is playing in his time, or if the move is valid, detecting and signaling endgame situations.

```
private void handle_PLAY(int p, char rplayer, int row, int col);
```

**TASK 4) Control the maximum time a player has to send PLAY**

To protect the game against players who take too long to play, you should create a timer in the *GameThread* class and manage it in order to exclude players who do not send a PLAY after *TTTServer.PLAY_TIMEOUT* milliseconds.

**TASK 5) Validate incoming messages and delete players playing out of order**

To make the game more robust to attacks, you must modify the functions that receive messages (*run*) in order to validate that they come from the registered IP address and port, and that they have the correct key, ignoring invalid messages.

You must also acount the number of times a player plays out of turn, excluding him when it fails for more than *TTTServer.MAX_FAULTS*. It is suggested that you modify the *PlayerInfo* class adding the necessary fields.

**TASK 6) Support multiple concurrent games in TTTServer server**

The version provided supports only one game at a time, having to restart the server to play another game. In this last task, you should remove this limitation. The changes will mainly affect the *TTTServer* class, which will have to create new objects of class *GameThread* each time a pair of new players join a game sending CONNECT messages. This change also requires creating a data structure with all active *GameThread* objects, so you can stop them when the server is shut down.

### 3.3. GOALS

ALL students should try to complete **at least the task 3**. A general introduction to the work is provided during the first week of the work, and you should to finish task 2. At the end of the second week you should have finished the task 4. At the end of the third and final week you should try to accomplish as much as possible, bearing in mind that it is better to do less and well (running and without errors), than everything and nothing working. For larger groups, it is suggested that the tasks (e.g. 2 and 1-3, 4-5 and 6) are done in parallel, shortening the total development time of the work.

## STUDENT POSTURE

Each group should consider the following:

- Do not waste time with the aesthetics of input and output data;
- Program in accordance with the general principles of good coding (using indentation for comments, using variables with names conform to its functions ...) and;
- Proceed so that the work is equally distributed to the two members of the group.