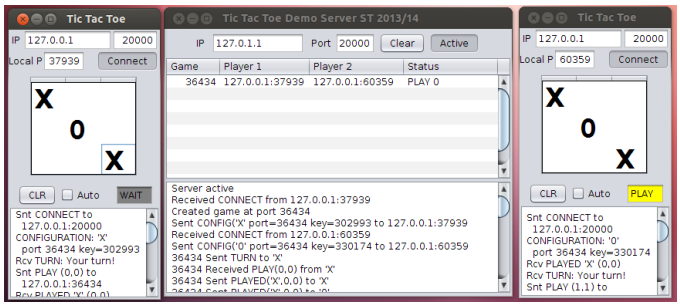
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Departamento de Engenharia Electrotécnica



Sistemas de Telecomunicações

2013/2014

Trabalho 1:
Aplicação sobre sockets – o jogo do galo

Mestrado integrado em Engenharia Eletrotécnica e de Computadores

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

Índice

1. OBJETIVO

Familiarização com o uso de *sockets* para comunicação entre máquinas e com o funcionamento da interface de programação *sockets* do Java.

O trabalho consiste no desenvolvimento de um jogo do galo distribuído em que os clientes comunicam com o servidor utilizando sockets datagrama.

> **Sugestões**: Em certas partes do enunciado aparece algum texto formatado de um modo diferente que começa com a palavra "Sugestões". Não é obrigatório seguir o que lá está escrito, mas pode ser importante para os alunos ou grupos onde ainda não haja um à-vontade muito grande com programação, estruturas de dados e algoritmia.

2. ESPECIFICAÇÕES

Pretende-se desenvolver um jogo do galo distribuído. O servidor do jogo comporta-se como um portal para os clientes dado que têm de conhecer apenas um endereço e porto de acesso. Cada jogo que estiver a decorrer tem um porto de acesso não conhecido publicamente. A comunicação entre os clientes (jogadores) e os servidores é realizada utilizando a troca de datagramas.

A ideia geral é a seguinte: os clientes acedem ao servidor de acesso (*portal*) para obter o endereço e porto do servidor de jogo (*game*), e uma chave (*key*) que terão de apresentar quando acederem ao servidor de jogo. O servidor de acesso vai lançar dinamicamente novos servidores de jogo, emparelhando os clientes dois a dois. O primeiro jogador associado joga como 'X' e o segundo como '0'. Cada servidor de jogo vai gerir todas as interações entre os dois clientes, indicando quando é a vez ("TURN"), recebendo e validando as jogadas ('PLAY') dos jogadores e enviando mensagens com a jogada realizada ('PLAYED'). Quando o servidor deteta o fim do jogo (quando um dos utilizadores faz três em linha, ou todas as posições do tabuleiro são ocupadas) envia a mensagem ('ENDED'), terminando nessa altura.

A aplicação cliente representa graficamente o estado do jogo e recebe os comandos do utilizador e executa todas as interações com o servidor de acesso e o servidor de jogo de forma a realizar o jogo do galo.

2.1. FLUXO DE MENSAGENS

A sequência de mensagens típica durante um jogo está ilustrada na figura 1, e consiste nas seguintes mensagens (assume-se que não existem erros do tipo de jogada inválida, chave inválida, erros na comunicação, etc.):

(1a e 1b) – Um jogador envia uma mensagem de pedido de ligação ("CONNECT") para o socket do portal (o serviço de acesso);

(2a e 2b) – O servidor de acesso processa o pedido de ligação e (2a) lança um novo servidor para o primeiro jogador, que joga como 'X'; ou, (2b) associa o jogador a um jogo existente, jogando como '0'. Neste segundo caso, arranca com um servidor de jogo, que inicia o algoritmo do jogo;

(3) – O servidor de jogo envia a mensagem "TURN" ao jogador 'X', sinalizando que é o próximo a jogar;

(4) – Após o jogador selecionar a próxima jogada, ele envia a mensagem "PLAY" com a posição selecionada "(0,0)" e aguarda;

(5a e 5b) – O servidor de jogo valida a jogada do jogador 'X' e envia uma mensagem "PLAYED" para os dois jogadores, a indicar a última posição jogada. De seguida ...

(6) – passa o jogo para o jogador '0' com uma mensagem TURN;

(7) – após o jogador '0' escolher a posição, gera uma mensagem PLAY, que será ecoada para os dois jogadores (8a) e (8b) se for válida;

A sequência (3)-(8) vai-se repetir até que um dos jogadores faça 3 em linha, ou se esgotem as posições. Admitindo que o jogador 'X' faz a jogada vencedora em (9), ...

(10a e 10b) – o servidor de jogo sinaliza o final de jogo e o resultado final através de uma mensagem END, que pode ter os valores "Won", "Lose", ou "Tie" em caso de empate.

Na comunicação com o servidor de jogo são usados *sockets datagrama*. Na realização do trabalho vai-se assumir que os pacotes não se perdem e que não há erros na comunicação, embora eles possam existir neste tipo de socket.

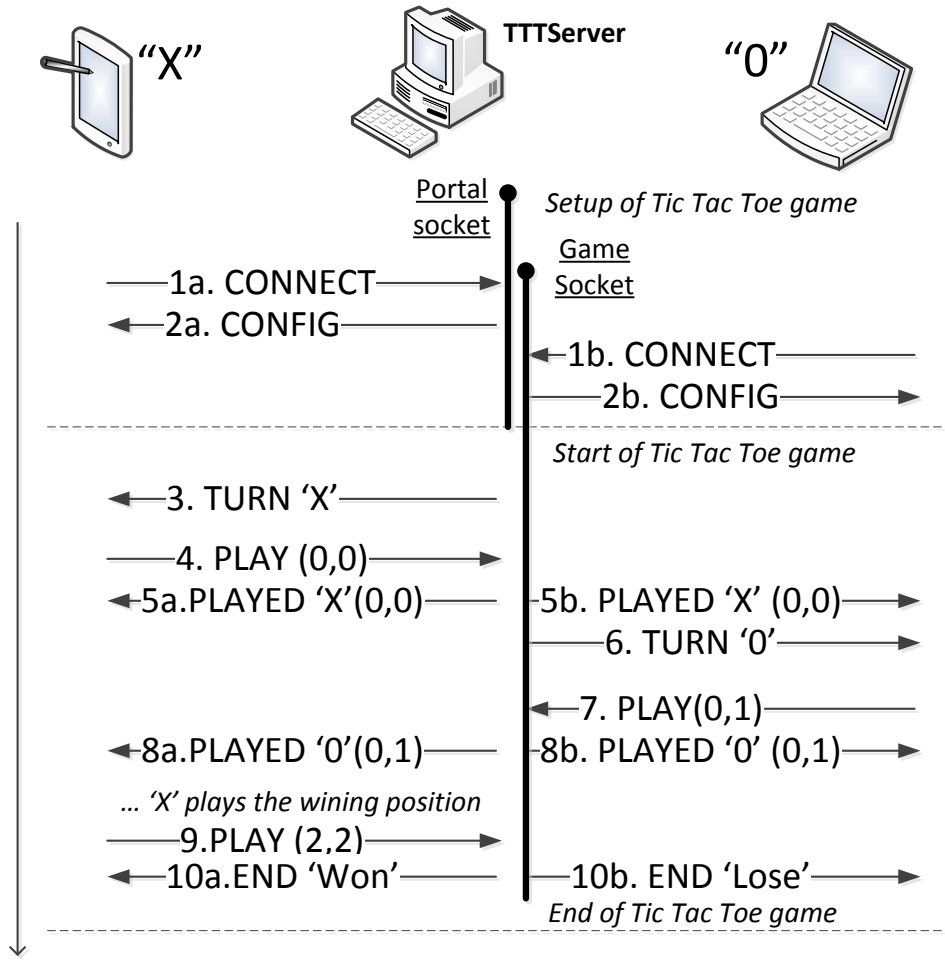
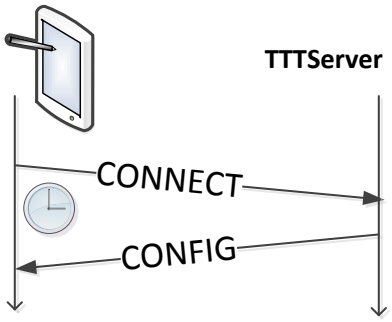


Fig 1 – Sequência de mensagens durante um jogo

2.2. SERVIÇO DE ACESSO (PORTAL)

O acesso ao serviço de acesso está representado nas mensagens 1a-2a e 1b-2b da Fig. 1, que obedecem ao seguinte protocolo de pedido-resposta:




```
Mensagem CONNECT:// sequência contígua de
 short type;  // Tipo de mensagem= 99
   // Use a constante TTTConfig.PCKT_CONNECT

Mensagem CONFIG: // sequência contígua de
 short type;  // Tipo de mensagem= 98
   // Use a constante TTTConfig.PCKT_CONFIG
 int key; // chave partilhada com o servidor
 char player; // jogador 'X' ou '0'
 int gamePort; // Porto do servidor de jogo
```

As mensagens **CONNECT** e **CONFIG** têm os parâmetros indicados. A mensagem CONFIG contém a chave aleatória *key* atrabuída pelo servidor de jogo, como é que o jogador joga (se inicia a partida *player='X'*, ou é o segundo a jogar *player='0'*), e *gamePort* contém o porto do servidor de jogo. O cliente usa um relógio (*timer*) para detetar quando a resposta do servidor se atrasa mais do que 10 segundo, terminando o jogo.

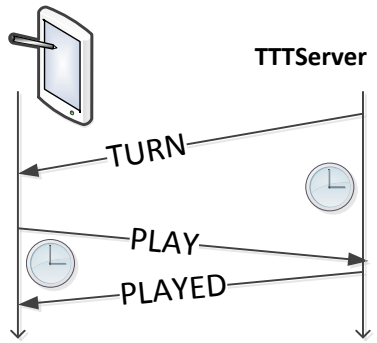
Como se disse, o *socket* do servidor de acessos é o único conhecido. Costuma-se chamar a isso ***well-known socket*** (o endereço IP é o da máquina respetiva e o que realmente é conhecido é o porto). Todos os outros *sockets* associados a jogos devem ter portos livres pois os seus valores são fornecidos aos vários participantes nos jogos. O porto do *socket* conhecido do primeiro servidor executado é o **20000**. Caso arranque mais do que um servidor na mesma máquina, estes terão os números 20001, 20002, etc.

Sugestões 1 – Vários jogos ativos

Numa primeira fase pode admitir que um servidor só suporta um jogo de cada vez (se lhe parecer que simplifica a programação). Caso disponha de tempo, deve preparar o servidor para suportar vários jogos em paralelo. Recomenda-se a utilização de uma lista para guardar os jogos ativos, permitindo terminar todos os jogos ativos. Não se esqueça que não deve em nenhuma circunstância bloquear a thread principal (associada à interface gráfica). Qualquer operação bloqueante (e.g. leitura num *socket*) deve ser realizada dentro de uma thread independente.

2.3. JOGO – JOGADAS

O acesso ao serviço de jogo usa *sockets datagrama*, sendo ilustrado pelas mensagens 3-5 e 6-8 da Fig. 1, que obedecem ao protocolo seguinte:




```
Mensagem TURN: //sequência contígua de
 short type;  // Tipo de mensagem= 10
   // Use a constante TTTConfig.PCKT_TURN
 int key; // chave partilhada com servidor
 char player; // jogador 'X' ou '0'

Mensagem PLAY: //sequência contígua de
 short type;  // Tipo de mensagem= 20
   // Use a constante TTTConfig.PCKT_PLAY
 int key; // chave partilhada com servidor
 char player; // jogador 'X' ou '0'
 byte row; // linha = 0, 1 ou 2
 byte col; // coluna = 0, 1 ou 2

Mensagem PLAYED: //sequência contígua de
 short type;  // Tipo de mensagem= 11
   // Use a constante TTTConfig.PCKT_PLAYED
 int key; // chave partilhada com servidor
 char player; // jogador 'X' ou '0'
 byte row; // linha = 0, 1 ou 2
 byte col; // coluna = 0, 1 ou 2
```

O servidor de jogo envia a mensagem TURN ao próximo jogador a jogar, aguardando um tempo máximo de 20 segundos pela jogada do jogador. O jogador envia a jogada através de uma mensagem PLAY que caso seja válida, é ecoada para os dois jogadores através de uma mensagem PLAYED. Todas as mensagens contêm a chave partilhada com o utilizador (*key*) e a identificação do jogador que joga (*player*). As mensagens PLAY e PLAYED contêm as coordenadas da posição da jogada (*row*, *col*).

Sugestões 2 – validação de jogada: O servidor de jogo pode validar se o IP e porto correspondem ao socket do jogador que enviou CONNECT e se está a jogar na sua vez.

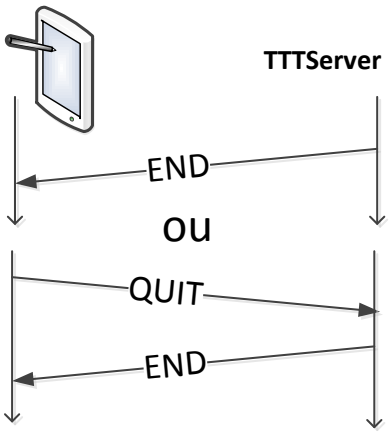
Sugestões 3 – número máximo de jogadas fora de vez: A aplicação cliente pode permitir que os jogadores joguem fora de vez. O servidor de jogo deve detetar este evento, e caso existam mais do que 4 jogadas fora de vez, o cliente deve ser punido com a perda do jogo.

Sugestões 4 – tempo máximo de espera: Se houver tempo, o servidor de jogo deve validar se o tempo máximo para enviar a mensagem PLAY é respeitado. Numa primeira fase de implementação, recomenda-se que essa validação não seja feita.

2.3. JOGO – FIM DO JOGO

O jogo termina quando um jogador marcar três posições em linha ou caso todas as posições sejam marcadas, sendo gerada uma mensagem END, ilustrada na mensagem 10 da Fig. 1, com a estrutura representada abaixo. A mensagem END não inclui o nome do jogador mas inclui o código de resultado *res* que indica se o jogador ganhou, perdeu ou empatou o jogo.

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




```
Mensagem END: //sequência contígua de
 short type;  // Tipo de mensagem= 90
   // Use a constante TTTConfig.PCKT_END
 int key; // chave partilhada com servidor
 byte res; // resultado para o jogador:
   // TTTConfig.END_GAME_WON= 1  - Ganhou
   // TTTConfig.END_GAME_LOSE= 2 - Perdeu
   // TTTConfig.END_GAME_TIE= 3  - Empate

Mensagem QUIT: //sequência contígua de
 short type;  // Tipo de mensagem= 97
   // Use a constante TTTConfig.PCKT_QUIT
 int key; // chave partilhada com servidor
 char player; // jogador 'X' ou '0'
```

3. DESENVOLVIMENTO DO PROGRAMA

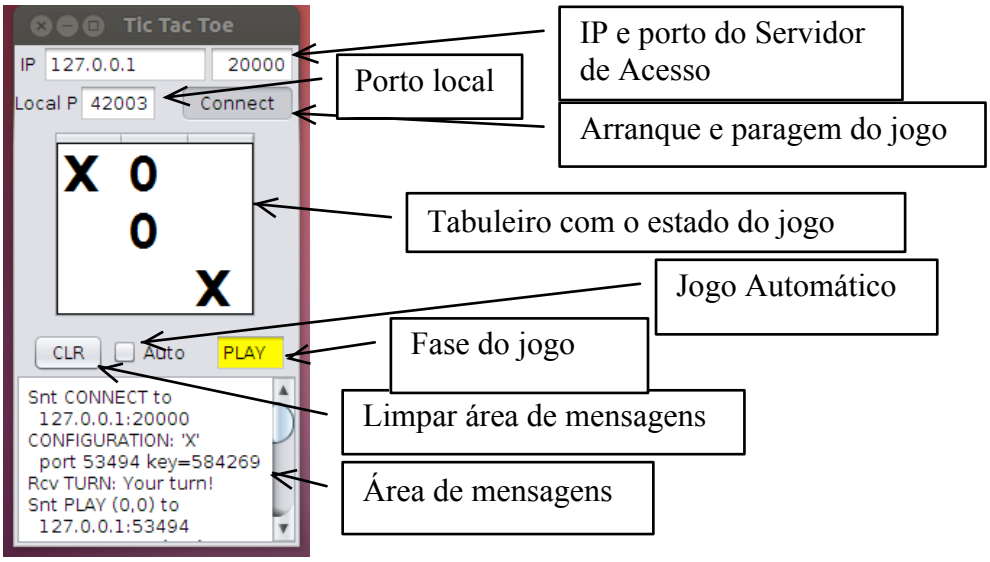
3.1. CLIENTE

A aplicação cliente é fornecida totalmente realizada juntamente com o enunciado do trabalho. Foi desenvolvida uma aplicação em Java com a interface gráfica representada em baixo. Quando se prime o botão "*Connect*" o cliente cria um socket datagrama com o porto

indicado em *Local P*, e comunica com o servidor de acesso no endereço IP e porto na primeira linha, realizando todos os passos representados na Fig. 1.

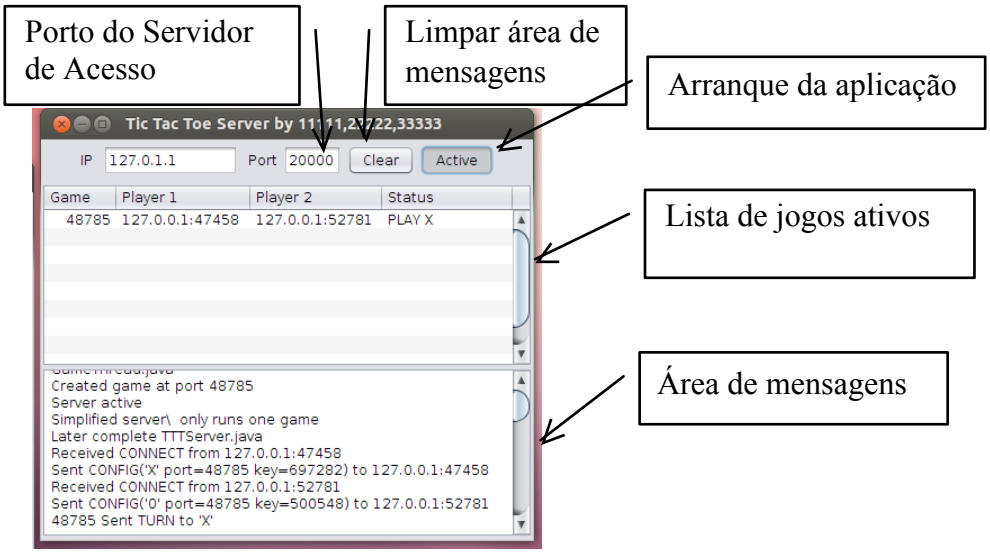
Na interface gráfica a aplicação cliente representa um tabuleiro com o estado do jogo e indica qual é a fase do jogo (i.e. se é a vez de jogar, ou se tem de esperar). O botão *Auto* seleciona se o cliente joga automaticamente ou se espera que o utilizador selecione as posições a jogar.

Pode correr o cliente a partir do terminal com o comando: `java -jar TTTClient.jar`



3.2. SERVIDOR

O trabalho consiste na realização do bloco servidor. Para facilitar o desenvolvimento do programa e tornar possível o desenvolvimento do programa durante as três aulas previstas, é fornecido juntamente com o enunciado um programa *TTTServStudents* incompleto, com a interface gráfica representada abaixo, que já realiza parte das funcionalidades pedidas. Cada grupo pode fazer todas as modificações que quiser ao programa base, ou mesmo, desenhar uma interface gráfica de raiz. No entanto, recomenda-se que invistam o tempo na correta realização da aplicação distribuída proposta neste enunciado.



O programa fornecido é composto por cinco classes em dois pacotes. O pacote *game* é comum ao cliente e ao servidor e tem as classes:

- *TTTConfig.java* (completa) – Classe com a configuração da aplicação que define constantes e funções estáticas;
- *GameState.java* (completa) – Classe que memoriza o estado do jogo, detetanto situações de vitória e o fim do jogo.

O pacote *server* implementa as funcionalidades do servidor. Este pacote tem as classes:

- *ConnectThread.java* (completa) – Thread que recebe pacotes no socket datagrama do serviço de acesso;
- *GameThread.java* (**a completar**) – Thread que realiza o serviço de jogo, guardando todos os dados de um jogo, e que recebe os dados no socket de jogo;
- *TTTServer.java* (**a completar**) – Classe principal com interface gráfica, que faz a gestão de sincronismo dos vários objectos usados.

O programa fornecido já realiza as três primeiras interações (1.-3.) da Fig. 1. Pretende-se que os alunos completem o código de forma a realizar todas as funcionalidades descritas na seção 2 deste enunciado. O código fornece um conjunto de classes e métodos destinados a suportar as realização das funcionalidades, que são descritas de seguida.

3.2.1. Classes e métodos de suporte

Classe *TTTConfig*

A classe *TTTConfig* do pacote *game* é usada para agregar todas as constantes e funções associadas usadas no protocolo. Define os tipos de mensagens e os tipos de resultado enviado na mensagem de fim de jogo, introduzidos anteriormente na secção 2. Para além disso, define dois métodos auxiliares, que permitem escrever de uma forma legível o valor de um tipo ou de um resultado:

```
/** 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);
```

Classe *ConnectThread*

A classe *ConnectThread* realiza a receção de pacotes datagrama para o serviço de acesso. Esta classe é muito semelhante à classe *Daemon_udp* apresentada na segunda aula do trabalho 0. Neste caso a thread descodifica o tipo da mensagem, apenas se invocando o método *handle_CONNECT_packet* da classe *TTTServer* quando se recebe o tipo *TTTConfig.PCKT_CONNECT*.

Classe *GameState*

A classe *GameState* memoriza o estado de um jogo, e permite detetar se o jogo já acabou, e determinar quem é o vencedor, ou se há um empate. Com a operação *play*, permite validar se cada jogada individual é válida. Desta forma, vai ser usado um objeto desta classe para cada jogo ativo.

```
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();
  }
```

Classe *TTTServer*

A classe *TTTServer* está quase toda realizada, oferecendo um conjunto de funcionalidades.

A interação com a interface gráfica é realizada através do seguinte conjunto de funções. Inclui uma função para ecoar mensagens para o caixa de texto e para controlar a tabela de jogos ativos:

```
  // 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
```

A classe o método *send_CONFIG*, que envia uma mensagem *CONFIG*, que usa o endereço IP e porto que está configurado no parâmetro *dp* para definir o destino do pacote.

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

A classe também define algumas constantes, como o tempo máximo à espera de uma mensagem *PLAY* depois de enviar *TURN* (*TTTServer.PLAY_TIMEOUT* em ms) e o número máximo de jogadas fora de ordem toleradas (*TTTServer.MAX_FAULTS*). A maior parte das restantes funções vai requerer pequenas modificações, para quem realizar todos os exercícios.

Classe *GameThread*

A classe *GameThread* realiza o servidor do jogo e mantém toda a informação relativa ao estado de um jogo. É uma thread baseada na classe *Daemon_udp*, que cria o socket *ds* localmente e trata localmente os pacotes recebidos.

Uma parte relevante desta classe é a gestão da informação relativa aos jogadores. Com esse objetivo, define a classe interna *PlayerInfo* e define um array de duas posições onde guarda as

informações sobre o jogador 'X' (na posição 0) e sobre o jogador '0' (na posição 1). Para além disso, define o seguinte conjunto de funções para obter e modificar as informações:

```
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);
```

Esta classe contém métodos completamente implementados para enviar mensagens TURN e END para um dos jogadores, mas a receção das mensagens e o envio da mensagem PLAYED estão incompletas.

```
// 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)
```

A classe já suporta os métodos para realizar o arranque e terminação do jogo, existindo a variável *active* que indica se o jogo está ativo. Note que um jogo não está ativo quando apenas está um jogador ligado, mas esse jogador pode desistir de esperar pelo início do jogo enviando uma mensagem QUIT. O método *start_game* envia a mensagem 3 representada na figura 1. O método *stop_game* realiza o envio das mensagens 10a e 10b representadas na figura 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. Tarefas a realizar

O trabalho está organizado numa sequência de tarefas:

TAREFA 1) Receber a mensagem PLAY

O método *run* da classe *GameThread* já descodifica parcialmente os pacotes recebidos, mas não descodifica completamente a mensagem PLAY. A primeira tarefa é completar esta função de maneira a ler os campos *row* e *col*, e invocar de seguida a função que trata a mensagem PLAY.

TAREFA 2) Enviar a mensagem PLAYED

Deve programar o método *send_PLAYED* da classe *GameThread* de maneira a preparar e enviar uma mensagem PLAYED.

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

TAREFA 3) Realizar a lógica do jogo

Esta tarefa está dividida em dois passos: Primeiro é necessário definir e inicializar campos (variáveis) na classe *GameThread* que permitam guardar o estado do jogo e memorizar quem é o próximo a jogar; num segundo passo é necessário programar o método *handle*_PLAY na mesma classe, de maneira a realizar a lógica do jogo. Deve validar se o jogador está a jogar na sua vez, ou se a jogada é válida, detetando e sinalizando situações de fim de jogo.

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

TAREFA 4) Controlar o tempo máximo que um jogador tem para enviar PLAY

Para proteger o jogo contra jogadores que demoram demasiado tempo a jogar, deve criar um temporizador na classe *GameThread* e deve controlá-lo de forma a excluir os jogadores que não enviarem um PLAY depois de *TTTServer.PLAY_TIMEOUT* milissegundos.

TAREFA 5) Validar as mensagens recebidas e excluir jogadores que jogam fora de ordem

Para tornar o jogo mais robusto a ataques, deve modificar as funções onde recebe as mensagens (*run*) de maneira a validar se vêm do endereço IP e porto registado e se têm a chave correta, ignorando as mensagens inválidas.

Deve também contabilizar o número de vezes que um jogador joga fora da sua vez, excluindo-o quando ultrapassar *TTTServer.MAX_FAULTS* vezes. Sugere-se que modifique a classe *PlayerInfo* acrescentando os campos que considerar necessário.

TAREFA 6) Suportar vários jogos concorrentes num servidor TTTServer

A versão fornecida só suporta um jogo de cada vez, tendo de se reiniciar o servidor para jogar mais um jogo. Nesta última tarefa, pretende-se que se levante esta limitação. As modificações vão afetar principalmente a classe *TTTServer*, que terá de passar a criar e arrancar novos objetos da classe *GameThread* cada vez que um par de novos jogadores se associar com mensagens CONNECT. Mas esta modificação também obriga a criar uma estrutura de dados com todos os objetos *GameThread* ativos, de forma a os poder parar quando se desligar o servidor.

3.3. METAS

TODOS os alunos devem tentar concluir **pelo menos a tarefa 3**. Na primeira semana do trabalho é feita uma introdução geral do trabalho, devendo-se terminar a tarefa 2. No fim da segunda semana devem ter finalizado a tarefa 4. No fim da terceira e última semana devem tentar realizar o máximo possível, tendo sempre em conta que é preferível fazer menos e bem (a funcionar e sem erros), do que tudo e nada funcionar. Para grupos mais numerosos, sugere-se que várias tarefas sejam realizadas em paralelo (e.g. 1-3 e 2; 4-5 e 6), encurtando o tempo de desenvolvimento do trabalho.

POSTURA DOS ALUNOS

Cada grupo deve ter em consideração o seguinte:

- Não perca tempo com a estética de entrada e saída de dados
- Programe de acordo com os princípios gerais de uma boa codificação (utilização de indentação, apresentação de comentários, uso de variáveis com nomes conformes às suas funções...) e
- Proceda de modo a que o trabalho a fazer fique equitativamente distribuído pelos dois membros do grupo.