ALGONQUIN COLLEGE

ASSIGNMENT 3 – Piccross[1] in Networking

General View

Due Date: prior or on **Week 10 - Dec 11th 2021 (midnight)**

- **Attention:** Because it is the end of the semester, we do not have a 2nd due date.

Earnings: 15% of your course grade.

Purpose: Create the networking service for Piccross.

- This is the third assignment in JAP. We will adapt the "Piccross" game developed before to use some features of the networking.
- **PART I:** Server definition: We need to use one specific server
  - The server must be able to work with multiple clients.
  - Server is supposed to reply clients.
- **PART II:** Client definition: What you need to include to update the previous GUI:
  - Using a new configuration received from server.
  - Send / receive messages (name, movements, marks)

Part I – Updating the Piccross Project (C/S version)

1.1. USING C/S ARCHITECTURE

The new version of **Piccross** (also based on "Picross" game – see, for instance http://picross.net/) will require networking connection. Basically, we need to have:

- **GameServer class**: That is responsible for starting the ServerSocket and to perform communication with clients.

[1] The name "Piccross" is intentionally changed in comparison with the original "Picross" game since some functionalities will be different. This idea was originally proposed by Prof. Daniel Cormier, instructor in some labs this semester.

- **GameClient class**: Once connected by Socket this client is responsible to start a game with a defined configuration.

The idea is that the game (client) can work similarly with the previous MVC model, but some actions can interact with the server.

- Basically, these methods are:
  - Basic connection (including disconnect);
  - Creation of a new **random** game (to be sent to server);
  - Initial execution using the server game configuration;
  - For ***bonus***: record of all points and times from players.

1.2. SERVER COMPONENT

The interface of the server is very simple (remember that most of time, server programs are executed in the backend of applications and they do not have GUI interfaces).

Preview (remember that you are free to define different interfaces):

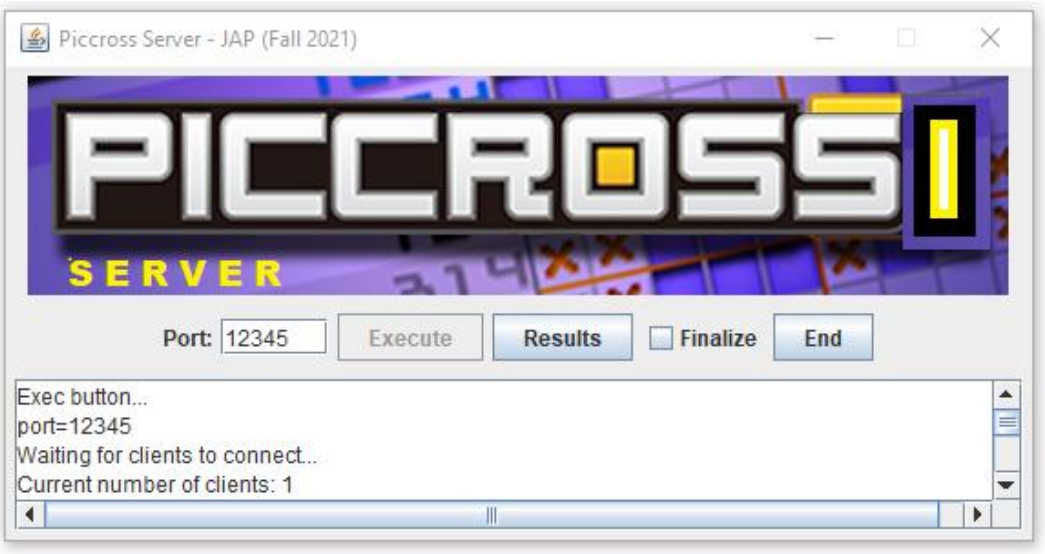

Figure 1: Server basic interface

Basically, here are the functions:

- You are able to define the **port** to be used;
- It is possible to **show all messages** (received from clients or logs) in a text area – this is replacing console messages.
- You can also **finalize** the server explicitly or implicitly (when there are no more clients to serve).

- **TIP**: If the check button “Finalize” is selected, when the last client ends, the server also finish (otherwise, the server continues waiting eventual new clients).

Note 1: About the Server
In this assignment, the server is providing basically “passive” and “volatile”, just providing the basic configuration for the game. However, the better implementation should consider correct persistence mechanisms (ex: using databases).

1.3. CLIENT COMPONENT

The client interface is also very simple also: before the game itself, the client is supposed to ask for connection and then, it can receive the send/receive data from/to server.

Preview (you can adapt this interface):

Figure 2: Client basic interface

The client functions include:

- Initial **connection** to host (you need to pass hostname and port);
- Creation of a **new random game** (see the definitions given in previous assignment). Ex: “01111,11100,10111,10110,01110”
- The operation to **send** of the configuration to the server.
- The operation to **receive** game configuration from the server.
- The execution (**play**) of the game using the configuration in the server.

- During execution, all messages can be shown in a text area (equivalent to log).
- You can also **finalize** the server explicitly (when there are no more clients to serve).
- ***Bonus***: Send data about player (name, points and time execution game) to the server.

Note 2: About the Client
In this assignment, the client will start with a basic interface and, once connected with the server, you can access the game itself (previous assignment).

1.4. PROTOCOL

Our networking solution is very simple. However, we need to be able to define a **minimum protocol** defining rules. Here is one example:

The send of each message (from client to server) is suppose to obey a proper format (that can be freely defined). For example:

- **Format used[2]:** **<Client_Id><Separator><Protocol_Id>[<Separator><Data_Using_Separator>]**.
  - **Protocol Sample 1 (Ending execution)**:
    - **String format for sending**: "**1#P0**"
    - **Expected result**: (Nothing – the client is supposed to close)
    - **Explanation**: This is the first action that any client can do: they can simply finish the game. The procedure is to inform the server that you are ending the connection. So, it is enough to send your identification[3] (1), in a new protocol identifier (P0)[4].
  - **Protocol Sample 2 (Sending game configuration to server):**
    - **String format for sending**: "**1#P1#00001,10111,00110,11111,00011**"
    - **Expected result**: (Nothing)[5]
    - **Explanation**: In this case, the client 1 is sending the message (P1) about the configuration of the game created previously (see the functionality to create a new random String that can set a new game configuration. Ex: 00001,10111,00110,11111,00011.

[2] The protocol present is not the unique way to define the communication between clients and server. However, if they are defined differently, it is impractical to test using the same server.

[3] The identification is the first answer received by a client when it has stablished connection with the server.

[4] This number is not sequential because it can be used as a special case to end the communication.

[5] I am simplifying the possibility of using responses from server (that could be simply the "echo" from original message), in order to make the communication simpler and clearer.

- **Protocol Sample 3 (Receiving game configuration from server)**:
  - **String format for sending**: Sending user name "1#P2"
  - **Expected result**: A game configuration. For example:

    "1#00001,10111,00110,11111,00011"
  - **Explanation**: After client request, server is returning a kind of string configuration. It can be generated by any other client that has send the previous command to the server.

- **Special cases:** Eventual exceptions during C/S communication can happen.
  - **Example 1:** During the **initial connection**, the client does not have any identification. So, the server is supposed to return the client id as the first message. Only after this, the client can provide its own identification.
  - **Example 2:** During the final connection, the better practice is to send a message to server that can finalize one specific thread used by this client. In this case, since the client is closed, the server does not return any other information[6].

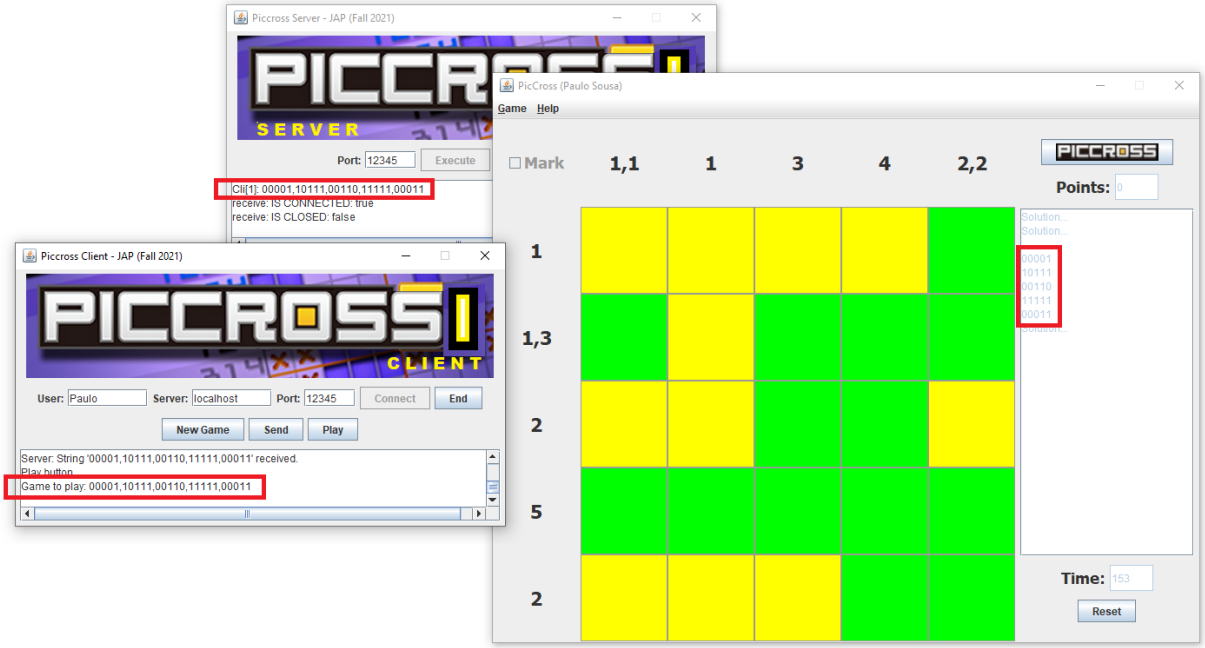

Figure 3: Example of protocol for game configuration.

It is important to emphasize that during almost all messages between client and server, the identification must be provided and then, correct updates can be done, especially considering the scenario for multiple clients.

[6] This final message is the better moment to update data from client (mentioned in the ***bonus*** task).

Note 3: About the Protocol
Is it really necessary to define **protocols** for different actions? Basically, it depends on the complexity of the service. If you just need to provide one kind of information (ex: a binary String), you can use only one single kind of message. However, in order to assist multiple clients, the identification is required to be included.

SERVER-SIDE

1. Our server (and client) start showing the basic interfaces previously described.
2. When the server starts the execution, it waits for client requests (there is a blocking process in the "accept()" function).
3. When a new client enters, a new thread must be associated to it. It means that the server is in condition to attend several clients simultaneously.
4. The first action is to send a number (id) to the client. It is important because when a client is sending any message using the protocol (as previously proposed), its identification is included in the beginning of the data.
5. Finally, the main action is supposed to be a method where you are developing actions to reply requests from different clients.

CLIENT-SIDE

1. When client starts, in the interface, it is necessary to inform the server location and port to be used (**default values** are shown: "localhost" and "12345").
2. Once done, the client can generate strings (using default dimension d=5 or eventually, random dimension) and transmit it to server.
3. After this, the client can start playing using the random configuration generated, using the **MVC solution** defined in the previous assignment.
4. Finally, the client can finalize, closing the socket connection (and eventually updating data to server – see ***bonus***).

C/S NOTES:

1. We can start executing the server or the client, but the connection just can have success if the server is active.

   ***TIP**: Be sure that host and port are free to be defined and matching during the execution.*

2. Use the SEPARATOR as a special char (ex: hashtag) to be used during communications.

   ***TIP**: Use String tokenizer to detect and separate parts of the message.*

3. Since the server and clients uses the GUI to execute C/S actions, be sure that buttons will not be **locked** when you perform these operations.

   ***TIP**: Use Threads (again, check the C/S demos)*.

4. When the client is finalizing, it is required to implement a dynamic behavior in the server such that, if it is the last client, the server can also finalize after receiving the connection close request.

   ***TIP**: Enumerate the current number of active clients and use some sentinels to check if the pool is empty*.

5. At any time, network connections errors are supposed to be managed by using pop-up messages.

   ***TIP**: Use ShowMessageDialogs*.

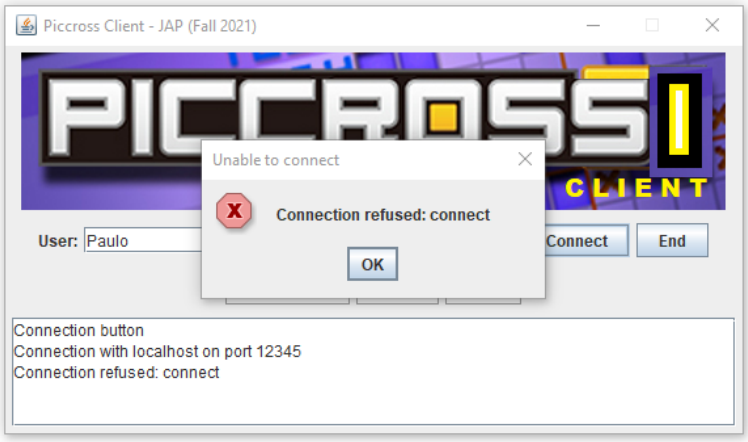

Figure 4: Basic example of error messages.

Part II – Final Notes

2.1. BONUS MARKS

Bonus marks will be given if the you can show the results send from different players during their execution. It means that clients must send the current marks and time for each game and the server can show in a kind of message. Eventually, it can be in the future, persisted in a kind of **database** (not done this time), that is typically used in the server side.

- ***TIP:** It is suggested to create a list (or Arraylist) of players – each one related to one specific client.*

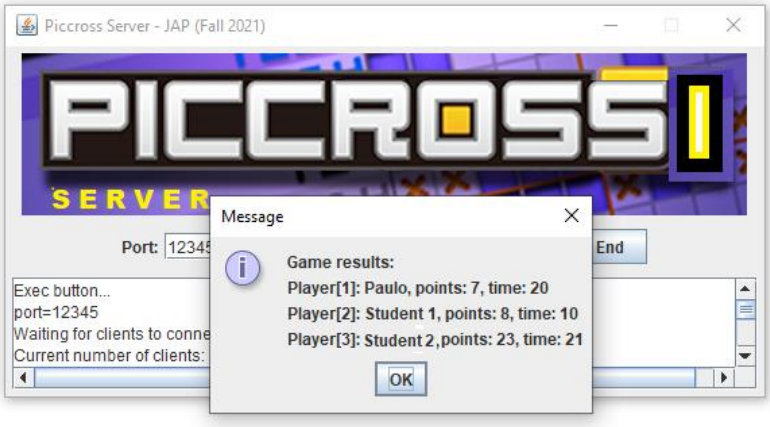

Fig. 5 – Showing results from several players in the game.

- ***TIP:** To develop this functionality, it is necessary to include a protocol from clients to server in the communication, where it is possible to transfer the points and time during one specific execution.*
  - **Protocol for Sending Data**:
    - **String format for sending**: "1#P3#Paulo#7#20"
    - **Expected result**: (Nothing)
    - **Explanation**: Suppose that you are sending the current data to the server. The procedure is supposed to provide the user data (for eventual records), sending to server: your specific identification (1), the protocol (P9) and include your data (name "Paulo", points "7" and time "20") – everything combined in one String[7].

Note 4: About the MVC Solution
Even though you can do modifications in the MVC solution to accommodate the new integration with the C/S architecture, it is NOT required. Good solutions can work independently. The way that you can test this later is to be able to execute the game inside and outside the client.

2.2. BASIC DOCUMENTATION

CODE ELEMENTS (*Basic criteria*)

- Code well organizes, methods and functions grouped logically, classes in order, etc.
- Label naming (variables, constants, methods, functions, etc.) consistent.

[7] It is possible to create protocols that send multiple lines. But here, we adopt the idea that using separators, we can combine all data using one single String and, therefore, reducing communications.

- Code consistently indented and spaced (define a consistency /organization).
- Good commenting which includes block comments.
- In the case of teams, Git files (or documents / images that can show the collaborative development) are required to be include.

JAVADOC (*following Java Standard*)

- Javadoc outputs are required, and the pages / folders must be included.

JAR (*Executable file*)

- Finally, it is required to create and include JAR files (that can be executed directly).

Note 5: About JAR
The C/S approach, as expected, is supposed to use two different files that are used in separate context (obviously, the client and server perspective). It means that commonly, we need to use separated JAR files for each case. However, it is possible to create a single main code that can execute both, and, in this case, one single JAR can be used.

2.3. STANDARD

❖ Please read the Assignment **Submission Standard and Marking Guide** (at "Assignments > Standards" section).

❖ **About Plagiarism**: Your code must observe the configuration required (remember, for instance, the "splash screen" using your name. Similarly, we need to observe the policy against ethic conduct, avoiding problems with the 3-strike policy...

2.4. RUBRICS

Marking Rubric

Maximum Deduction (%)	Deduction Event
up to 100	**Severe Errors:**
100	Late submission (due to final week due date, late submissions will not be accepted)
up to 100	Does not compile (using basically javac)
up to 100	Compile but crashes / freezes after launch
up to 100	**Does not comply with the problem specifications**
up to 50	The C/S does not treat multiple clients and have errors in dynamic execution.
up to 50	Protocol inconsistency (different strings connection)
up to 30	Exception handler not done (ex: connection problems) and treated by dialogs.
up to 20	Problems with GUI components (missing behaviors – ex: text area update).
up to 10	Inadequate comments (headers, code explanation), insufficient javadoc

up to 10	Missing or incorrect classes and package
up to 10	Other minor errors
up to 20	Bonus: Server solution can show results (points, time) from different players.
Final Mark	**Formula:** 15*((100- ∑ penalties + bonus)/100), max score 15%**.**

Submission Details

- **Digital Submission: Compress** into a **zip** file with all files (including source code, images, Javadoc, Jar).
- Upload the **zip** file on Brightspace. The file must be submitted prior or on the due date as indicated in the assignment.
- ***IMPORTANT NOTE:*** The name of the file must be **Your Last Name** followed by the last three digits of your student number followed by your lab **section number**. For example: **A11_Sousa123.zip**.
  - If you are working in teams, please, include also your partner. For instance, something like: **A11_Sousa123_Cormier456.zip**.
- ***IMPORTANT NOTE:*** **Assignments will not be marked if no source files are included.** All assignments must be successfully completed to receive credit for the course, even if the assignments are late.

File update: Nov 22nd 2021.

Good luck with A3 (Final Assignment)