Lab session 3

Exercise

Implement a networked application with client/server topology using the **TCP protocol.**

NOTE: A skeleton of the application is provided in the virtual campus.

The goal of this session is learning how to use the *select()* non-blocking mechanism when working with sockets, as many games need a high degree of interactivity (performing at least at 60 frames per second, for instance), and thus, they cannot afford waiting for sockets until they are available to read/write data from/to the network.

In this network topology, the **server** is typically waiting for new connections through al listen socket, and receiving and sending game data from/to clients through connected sockets. It is quite clear that working with TCP sockets, the server will need to maintain several simultaneous connections with clients (that is, several connected sockets). The specifics about the server implementation can be found in the class ***ModuleNetworkingServer*** files.

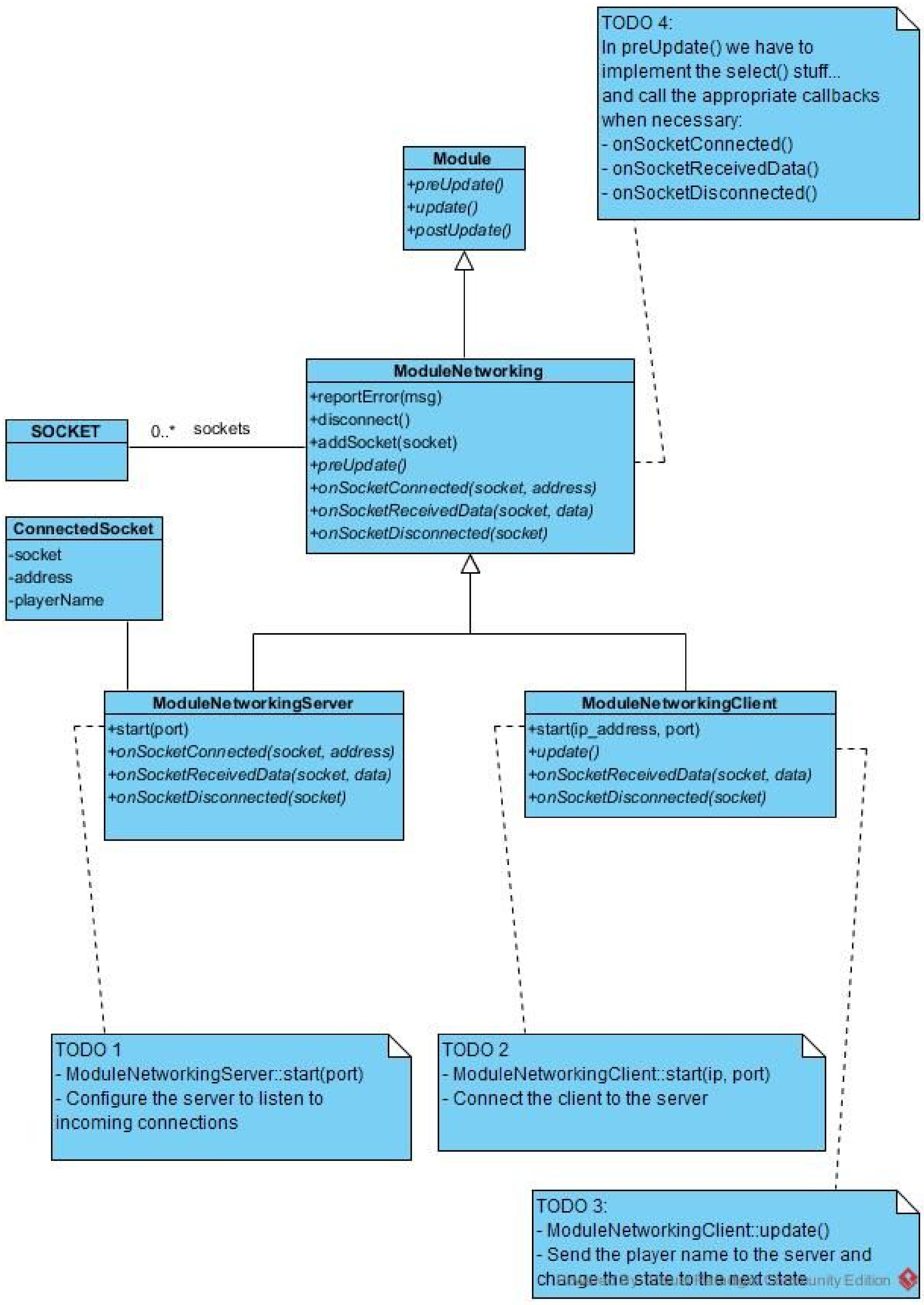
Regarding the **client**, the implementation will not be too different with respect to the previous' class exercise. The client will only work with a unique socket connected to the server. To see the specific code for the client part, refer to the ***ModuleNetworkingClient*** files.

You will see that both classes, ***ModuleNetworkServer*** and ***ModuleNetworkClient***, inherit from the common ***ModuleNetworking*** class. This base class provides the common mechanism to deal with sockets. Basically, we will implement the ***select()*** functionality in this class, to query which operations are available on each socket, and will notify about important events (new connections, incoming data, and disconnections) to the subclass using a few virtual methods. The subclasses will know what to do in each specific case, as they will need to handle those events differently.

The **tasks to implement** are roughly the following:

1. Make the server listen to incoming connections.
2. Make the client connect to the server.
3. Make the client send a “hello” packet with its player name.
4. Implement the select() functionality in order to dispatch events coming from sockets.

In the following figure, you can see the class diagram of the main components involved.


TODO 4:
In preUpdate() we have to implement the select() stuff... and call the appropriate callbacks when necessary:
- onSocketConnected()
- onSocketReceivedData()
- onSocketDisconnected()
Module
+preUpdate()
+update()
+postUpdate()
ModuleNetworking
+reportError(msg)
+disconnect()
+addSocket(socket)
+preUpdate()
+onSocketConnected(socket, address)
+onSocketReceivedData(socket, data)
+onSocketDisconnected(socket)
SOCKET
0..*
sockets
ConnectedSocket
-socket
-address
-playerName
ModuleNetworkingServer
+start(port)
+onSocketConnected(socket, address)
+onSocketReceivedData(socket, data)
+onSocketDisconnected(socket)
ModuleNetworkingClient
+start(ip_address, port)
+update()
+onSocketReceivedData(socket, data)
+onSocketDisconnected(socket)
TODO 1
- ModuleNetworkingServer::start(port)
- Configure the server to listen to incoming connections
TODO 2
- ModuleNetworkingClient::start(ip, port)
- Connect the client to the server
TODO 3:
- ModuleNetworkingClient::update()
- Send the player name to the server and change the state to the next state