# Tic-Tac-Toe Project Report

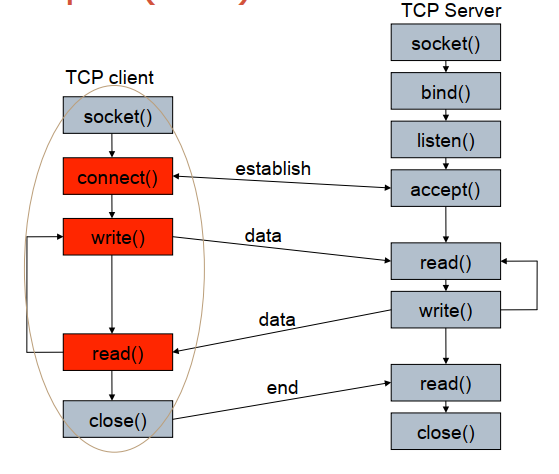
**Group 03: Bui Cong Thanh, Nguyen Quang Minh**

**Outline:**

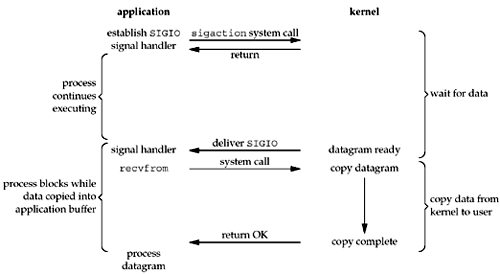
* Application description
* Game rule
* Application architecture
* Functionality
* Message design
* Working procedure
* Application evaluation

1. **Application description:** 
   * In this project, we develop the game Tic Tac Toe for 2 persons.
   * Player open a game client to play.
   * Login is required
   * Player need to wait for another player, first come first serve
   * Multiple clients connect to a server and transfer position data through that server
   * Programming language: C
2. **Game rule:** 
   * Two players, X and O, take turns marking the spaces in a 3×3 grid
   * The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row wins the game.
3. **Application architecture:** 
   * 1. **TCP client-server (Transmission Control Protocol)**

* Provide reliable communication
* Data rate control
* We use C socket to implement TCP client and server



* + 1. **Signal-driven I/O model**
* Use signals, telling the kernel to notify us with the SIGIO signal when the descriptor is ready.

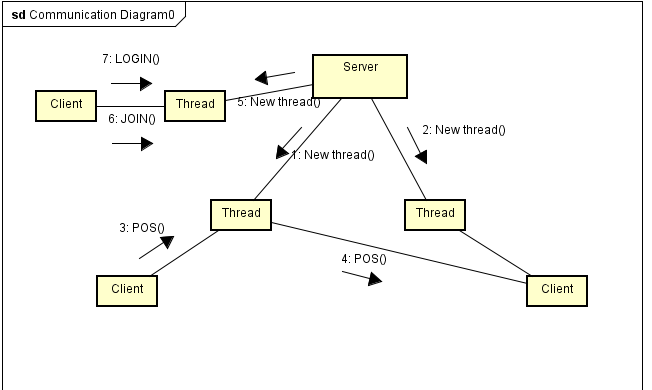


* We first enable the socket for signal-driven I/O and install a signal handler using the sigaction system call. The return from this system call is immediate and our process continues; it is not blocked. When the datagram is ready to be read, the SIGIO signal is generated for our process. We can either read the datagram from the signal handler by calling recvfrom and then notify the main loop that the data is ready to be processed, or we can notify the main loop and let it read the datagram.
* Regardless of how we handle the signal, the advantage to this model is that we are not blocked while waiting for the datagram to arrive. The main loop can continue executing and just wait to be notified by the signal handler that either the data is ready to process or the datagram is ready to be read.

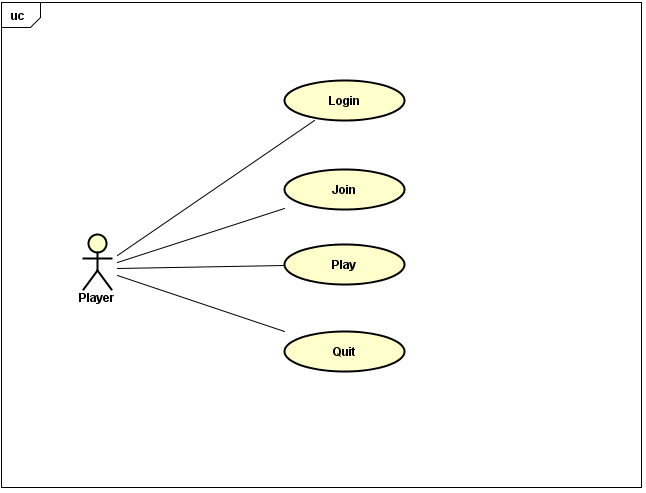
**Signal-Driven I/O for Sockets**

* To use signal-driven I/O with a socket (SIGIO) requires the process to perform the following three steps:
* A signal handler must be established for the SIGIO signal.
* The socket owner must be set, normally with the F\_SETOWN command of fcntl.
* Signal-driven I/O must be enabled for the socket, normally with the F\_SETFL command of fcntl to turn on the O\_ASYNC flag.
* Although setting a socket for signal-driven I/O is easy, the hard part is determining what conditions cause SIGIO to be generated for the socket owner. This depends on the underlying protocol.
  + 1. **Multithread TCP server**
* We use <pthread.h> library. For each connection to the server, it creates a thread to handle each client independently. We choose thread over forking server which uses process, because creating thread is cost less than creating process and it is easier to use shared variable between threads.

For example, in our projects, we have a player queue which is a shared variable. When user joins to a match, thread can easily access to player queue to enqueue and dequeue.

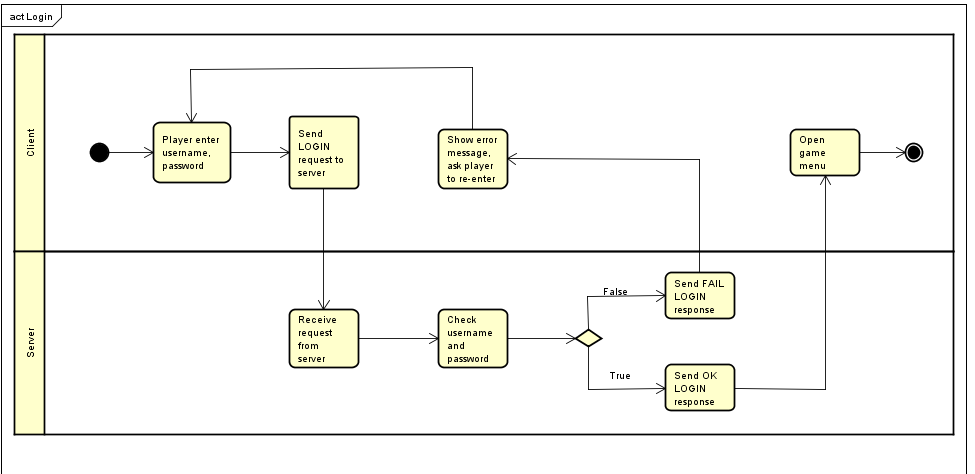
****

1. **Functionality:**

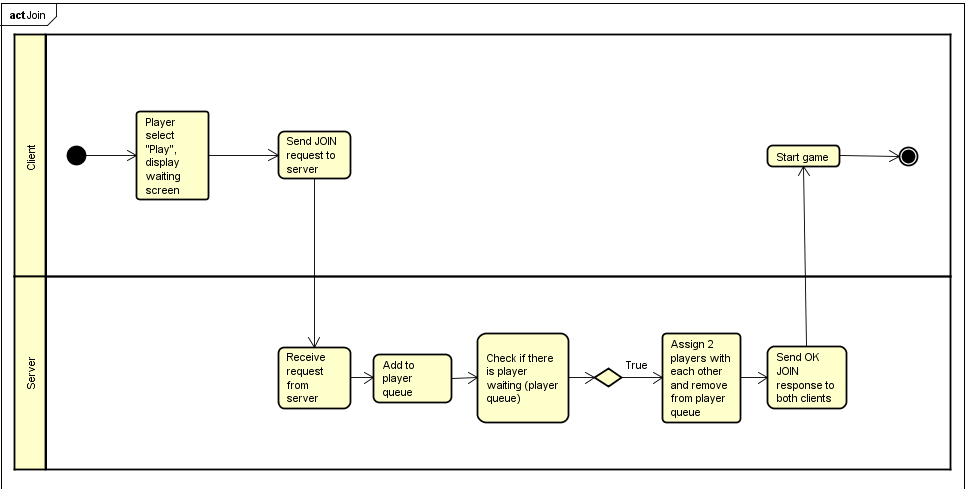


* User can login to play
* Join a game
* Play with each other in real time
* Quit

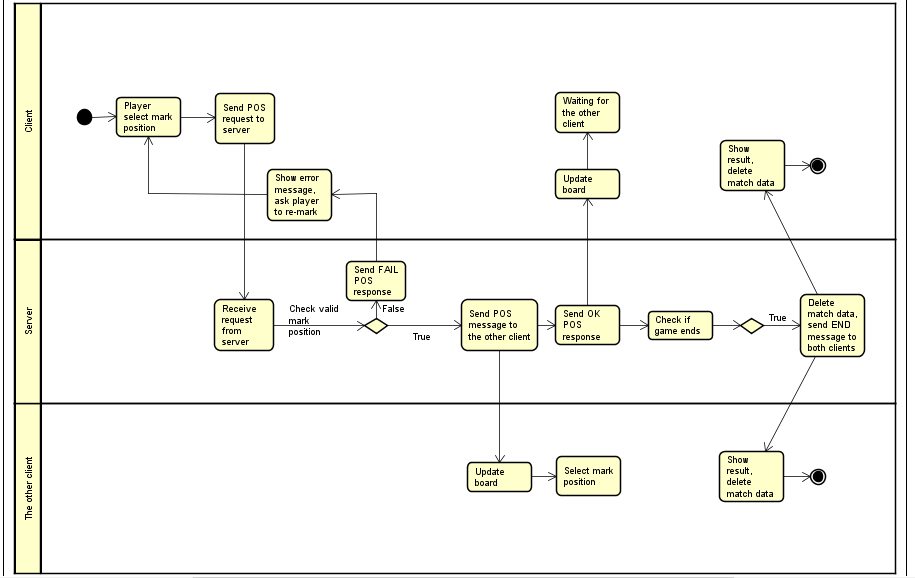
1. **Working procedure:**



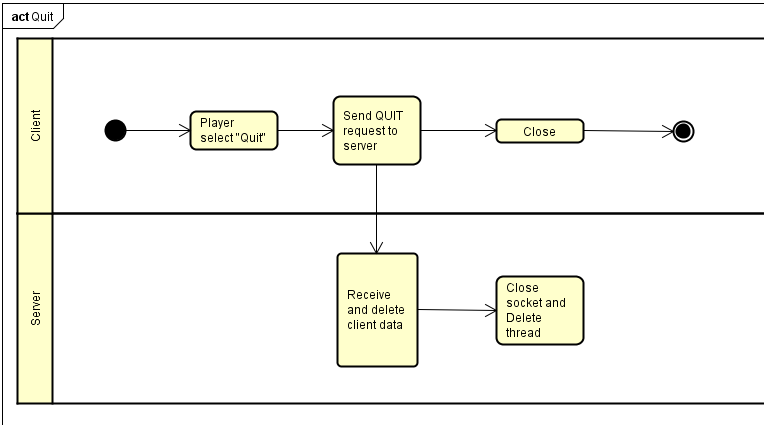
* Login: User is a client connect to server, user input name and password, server checks if user exists and login.



* Join: User select play. Server will wait for another user connecting and assign these 2 users to play with each other.



* Play: While playing, client send its mark position to server and receive the other client position from server.



* Quit game: Log out and close client.

1. **Message design:**

* Message formats:
  + Client to server:
    - LOGIN <username> <password>: Send username and password from client to server
    - JOIN: Send message from client to server that user selects JOIN
    - POS <x\_position,y\_position>: Send position from client to server
    - QUIT: Send message from client to server that user quits
  + Server to client:
    - OK <req> <message>: Send successful message from server to client

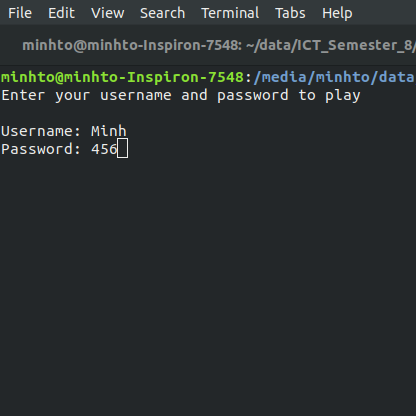
(req is request name of which this message responses, e.g. OK LOGIN logged)

* + - FAIL <req> <message>: Send fail message from server to client
    - POS <x\_position,y\_position>: Send position of a client from server to the other client
    - END <winner\_turn>: Send message to clients to tell that the match is finished
* Message sequences in communication: (details in figure above)
  + Login: LOGIN <username> <password> -> OK/FAIL LOGIN <msg>
  + Join: JOIN -> OK JOIN <turn>
  + Play: POS <position> -> … -> END <winner\_turn>
  + Quit: QUIT

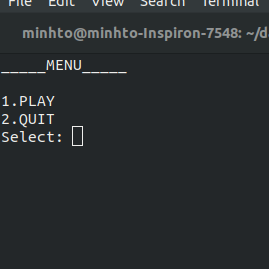
1. **Application evaluation**

Our application is able to run all the functions mentioned above:

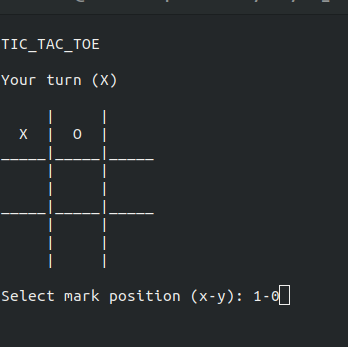
* Login:



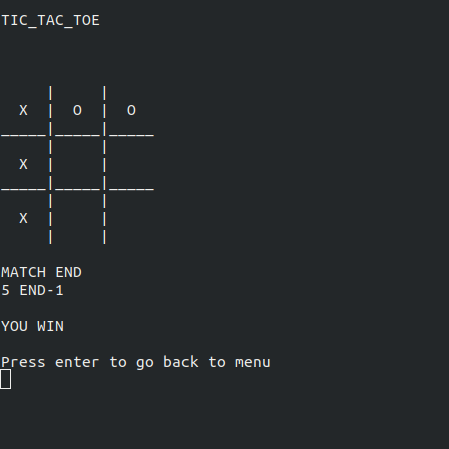
* Menu screen:



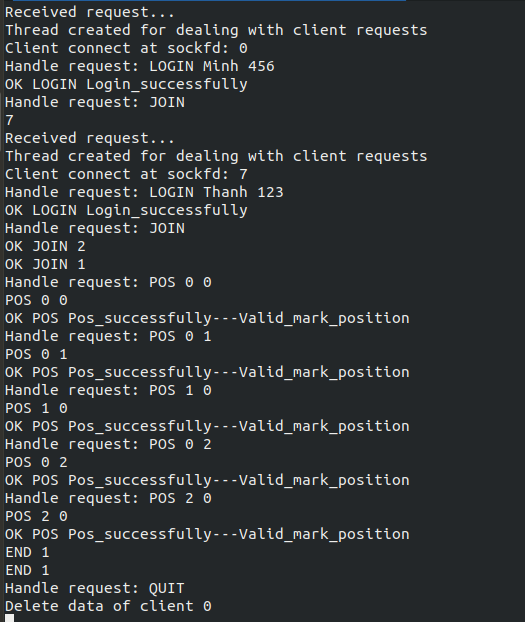
* Playing screen:



* Match end:



* Messages sent/received on server side:



* Advantages:
  + Application is able to run all the main functions mentioned above.
  + The multi-thread server can connect to many clients as expected.
  + Messages exchanged between server and client are as we have designed.
* Disadvantages:
  + User experience and security is not taken care of since this is not our main focus while doing this project.