Bora Sahin
Backgammon Client / Server Application Design

Abstract

The backgammon application design document for client and server was written as part of the term project effort, in partial fullfillment of SWE544 Internet Programming course requirements at Bogazici University.

It explains client-server communication, client and server architecture and design.

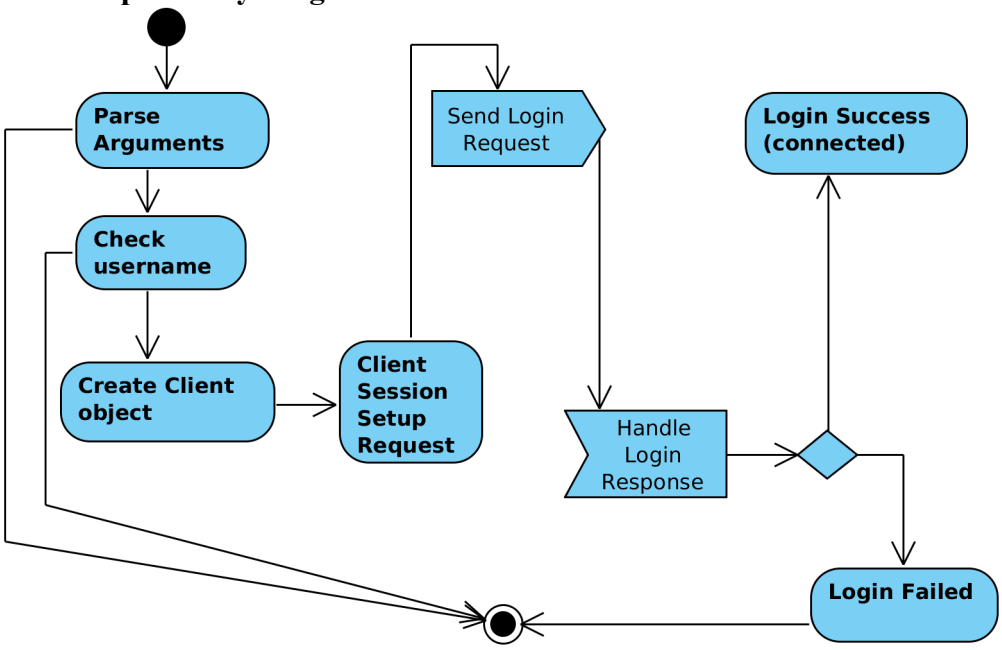
Table of Contents

1. Client / Server Communication

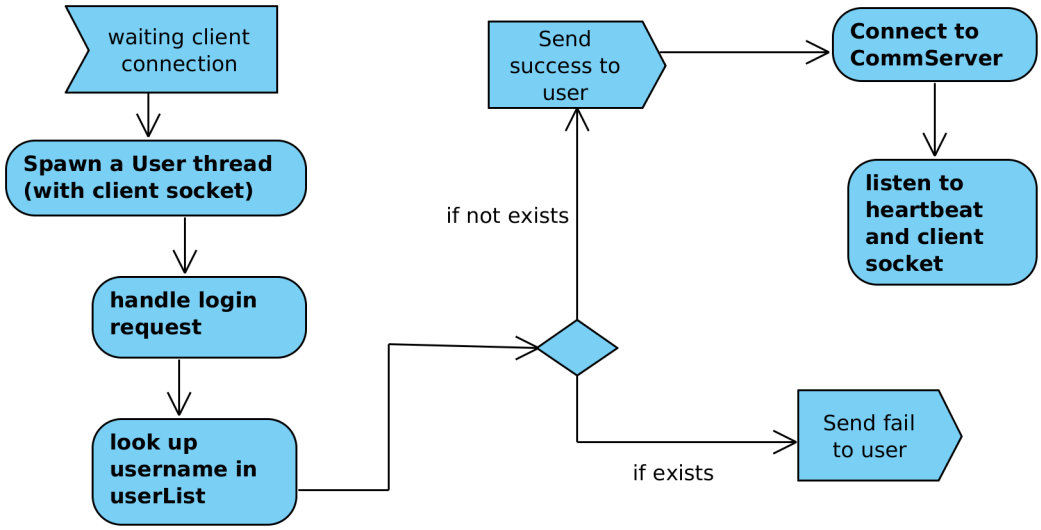
1.1. Session Setup

Client Messages	Server Messages
CLOGIN(username) ->	
	Server receives CLOGIN(username) and checks if that username is used to log in to the system
	<- SLRSPS(success)
	<- SLRSPS(fail)

Client Session Setup Activity Diagram



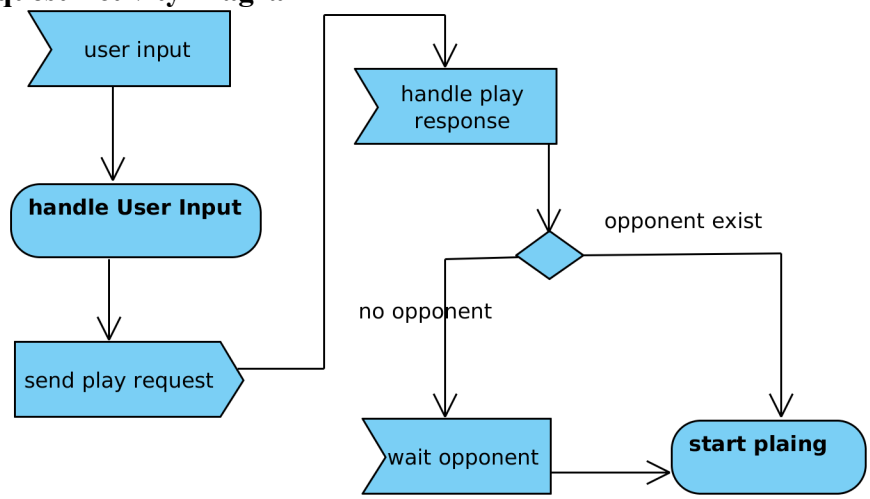
Server Session Setup Activity Diagram



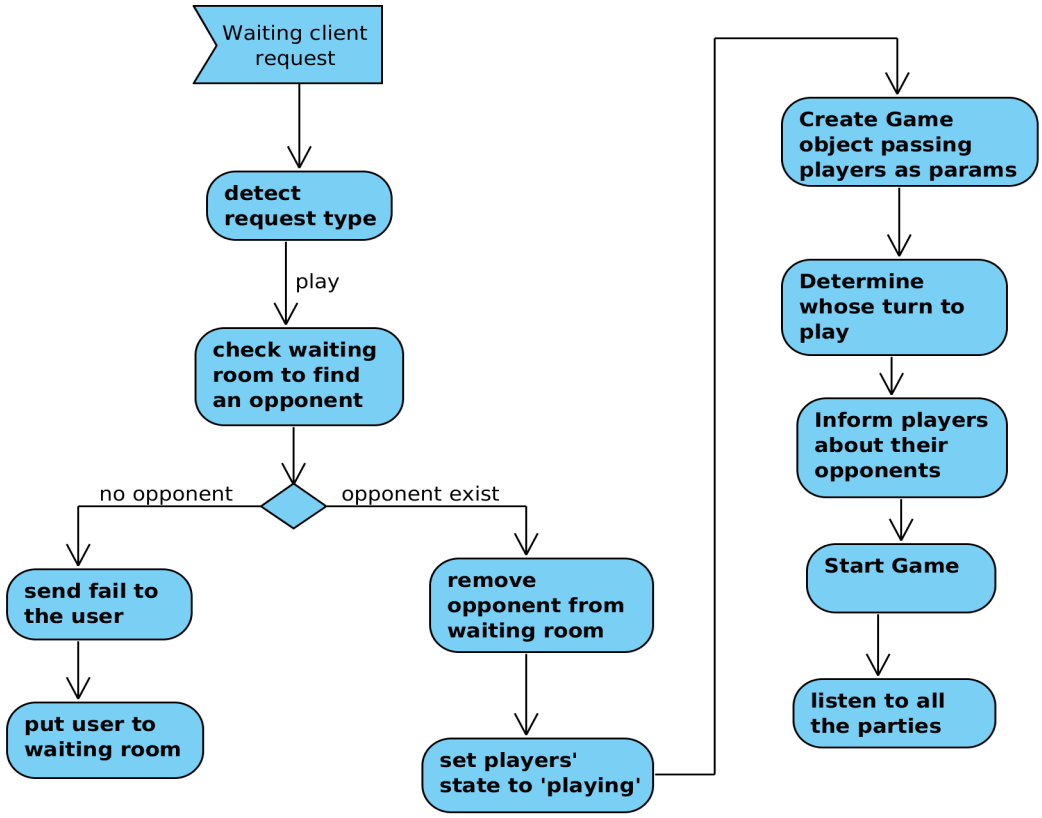
1.2. Play Request

Client Messages	Server Messages
CREQST(play)	
	Server receives CREQST(play) and checks if there is somone in the waiting room wanting to play
	<- SREQRP(play, success)
	<- SREQRP(play, fail)

Client Play Request Activity Diagram



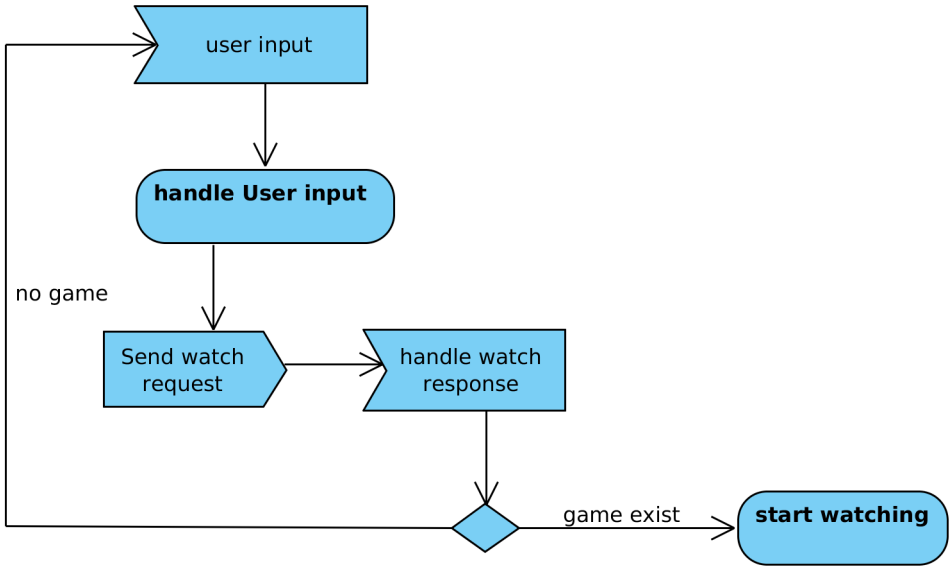
Server Play Request Activity Diagram



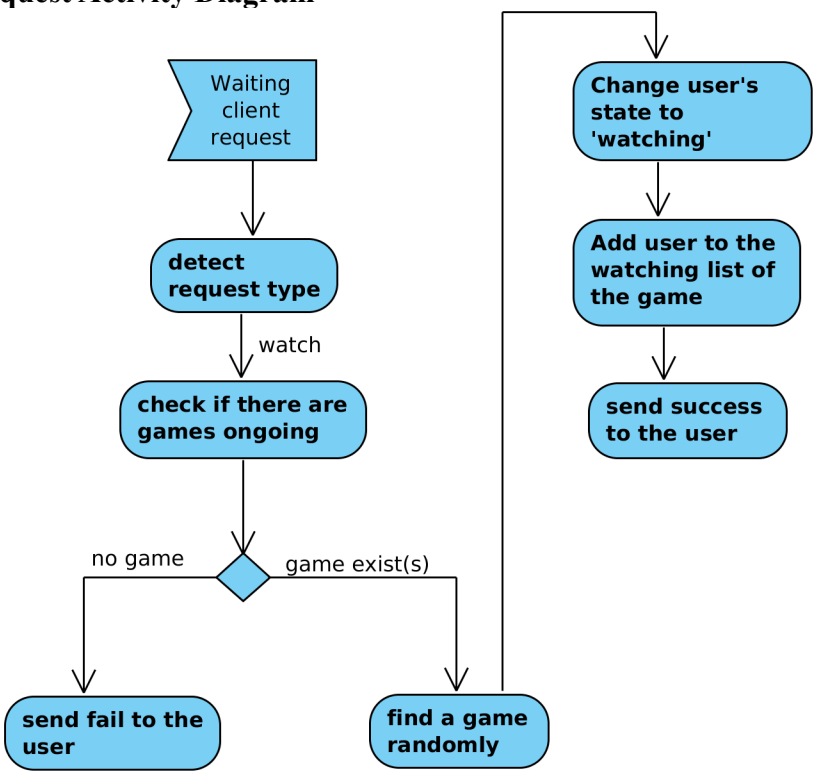
1.3. Watch Request

Client Messages	Server Messages
CREQST(watch)	
	Server receives CREQST(watch) and checks if there is a match or are matches. If there is, then attaches watcher to a game randomly
	<- SREQRP(watch, success)
	<- SREQRP(watch, fail)

Client Watch Request Activity Diagram



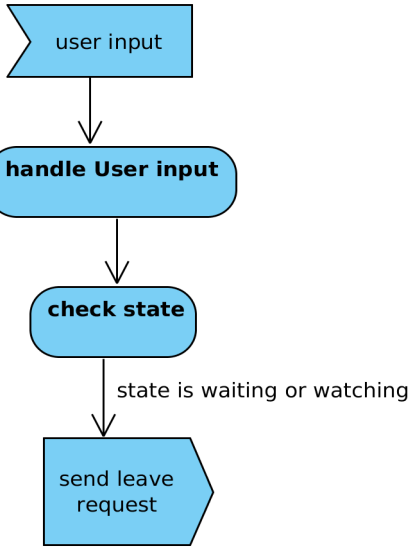
Server Watch Request Activity Diagram



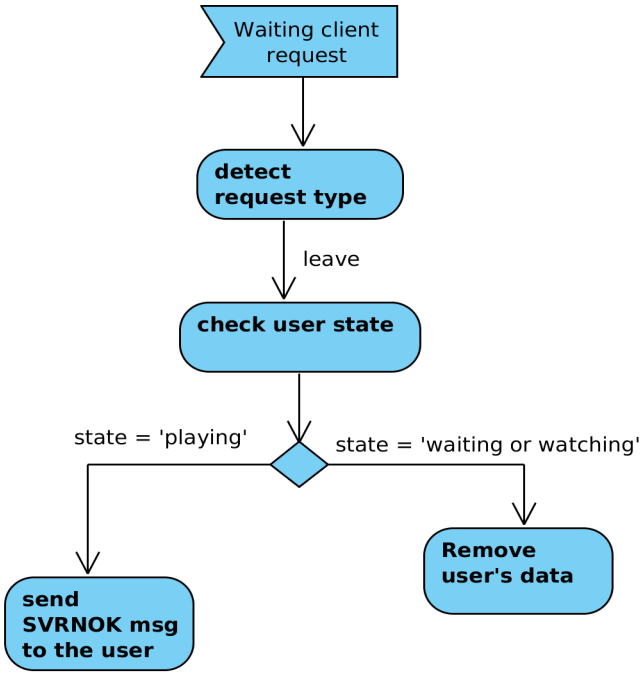
1.4. Leave Request

Client Messages	Server Messages
CREQST(leave)	
	Server receives CREQST(leave) and checks the state of the user. If it is either 'waiting' or 'watching' then remove its data and does nothing. If its state is 'playing', then SVRNOK is sent to the client
	nothing
	<- SVRNOK()

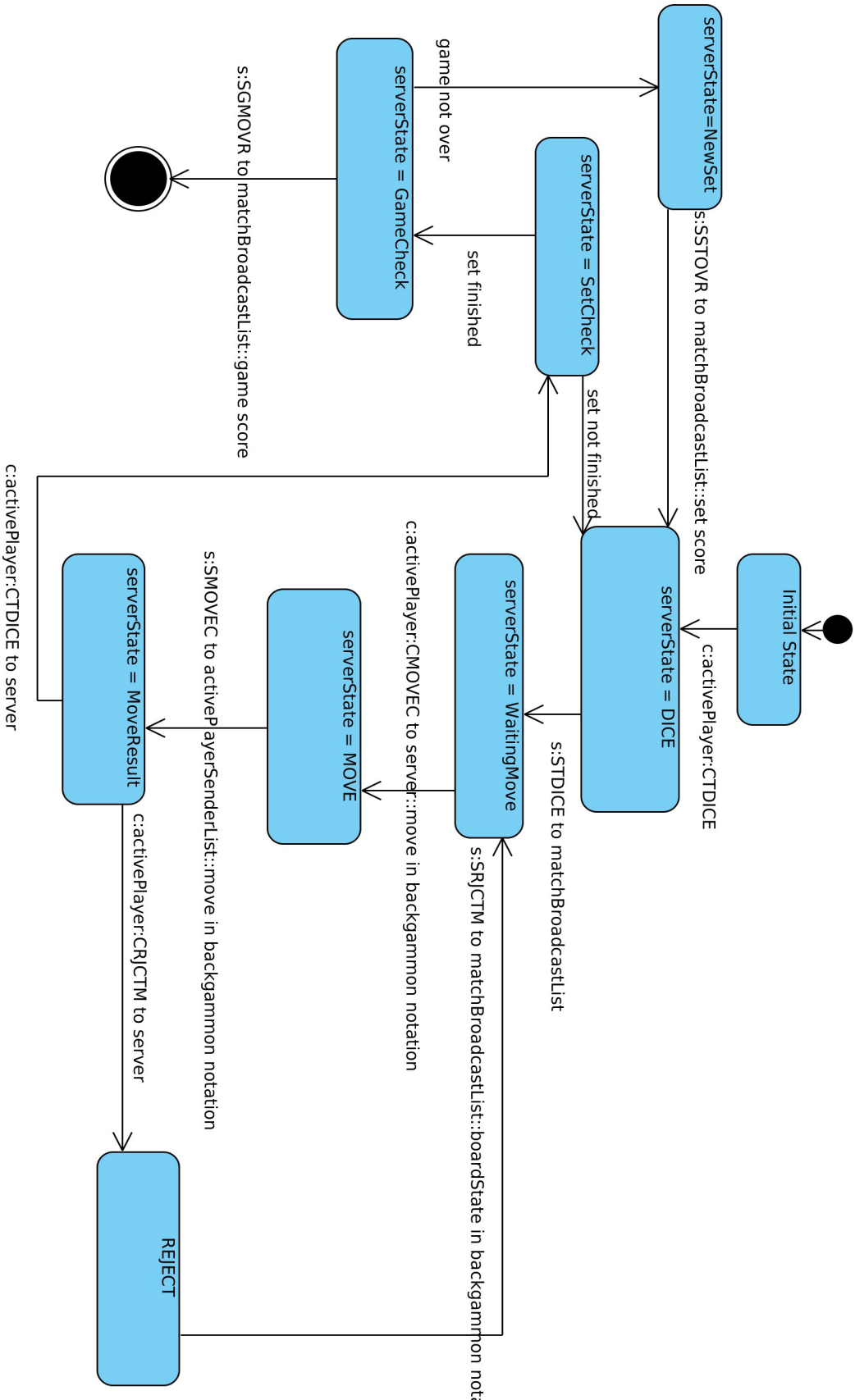
Client Leave Request Activity Diagram



Server Leave Request Activity Diagram



2. Playing State Machine and Messaging



Above state machine diagram represents the situation in a game occuring in the server, while also showing messages exchanged by server and players.

When two players who want to play backgammon are in the server, a game is started. The first thing which is done by the server is to determine whose turn to play and color for each player. Therefore, server throws single dice on behalf of players and informs them about the result (SREQRP). They are shown in previous sections. Before commenting furthermore, a few things need to be cleared:

- The player who wins to the challange has the right to request throw dice from the server and is marked as active player
- All the moves are represented in backgammon notation
- activePlayerSenderList represents all the users except the active player
- Passive player is the one who is waiting his/her opponent's move
- passivePlayerSenderList represents all the the users except the passive player
- watchersList represents the watchers of the game
- matchBroadcastList represents all the users including active and passive player and watchers
- c: refers to a message sent by the client
- s: refers to a message sent by the server

Active player starts to the game by requesting throw dice from the server, which is represented by CTDICE message (c:ActivePlayer:CTDICE). Upon receiving this message, server enters into 'DICE' state and throws dice, which are broadcasted to matchBroadcastList (s:STDICE) and brings the server state to 'WaitingMove'. Active player which is said to have the right to make a move decides what to play and tells it to server (c:ActivePlayer:CMOVEC). When server receives CMOVEC from a player, rebrands it as SMOVEC and then sends to activePlayerSenderList (s:SMOVEC). At this point activePlayer is changed to other player. Opponent gets SMOVEC and may have two reservation: Accept or Reject. If opponent accepts the move, it sends CTDICE request to the server (c:CTDICE); if not, then sends CRJCTM (c:CRJCTM). It can be described as below:

Scenario 1

Player1	Server	Player2
CTDICE ->	Player1:active, Player2:passive	
	Player1:active, Player2:passive <- STDICE ->	
Receives STDICE		Receives STDICE
CMOVEC ->		
	Receives CMOVEC and rebrands it as SMOVEC Player1:passive, Player2:active SMOVEC ->	
		Receives SMOVEC, then decides its fate
		ACCEPT

		<- CTDICE
	<- STDICE ->	

Scenario 2

Player1	Server	Player2
CTDICE ->	Player1:active, Player2:passive	
	Player1:active, Player2:passive <- STDICE ->	
Receives STDICE		Receives STDICE
CMOVEC ->		
	Receives CMOVEC and rebrands it as SMOVEC Player1:passive, Player2:active SMOVEC ->	
		Receives SMOVEC, then decides its fate
		REJECT
		<- CRJCTM
	Server receives CRJCTM and rebrands it as SRJCTM and sends it to matchBroadcastList Player1:active, Player2:passive <- SRJCTM ->	
Receives SRJCTM		Receives SRJCTM
Needs to make another move		

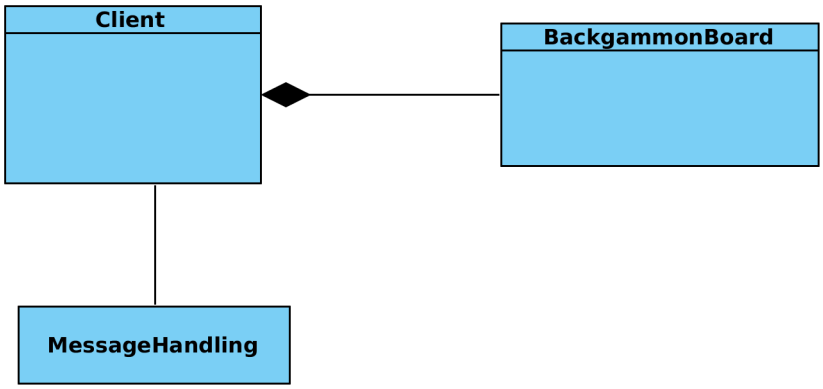
After a move is accepted by an opponent, it is told to the server by sending CTDICE message. As server keeps some backgammon logic, namely to tell if a game is won or not, it should check it. A game consists of sets and a game finishes if one player reaches 5 set point. When a set is won, server informs all the parties with a message called SSTOVR and then goes on to check if the game is finished or not, and if finished, then informs all the parties with a message called SGMVOR. At this point, sessions of all the parties are closed. SSTOVR is just an informational message. Its only impact is that the player who won the set has the right to request throw dice from the server.

If a player loses his/her connection to the server, which means two back-to-back PING messages missed by the client, then server informs all the parties that a player lost his/her connection by sending STEARD message. All the data related to users and game are removed from the server and it is expected that clients receiving this message should also act upon appropriately.

3. Client Architecture and Design

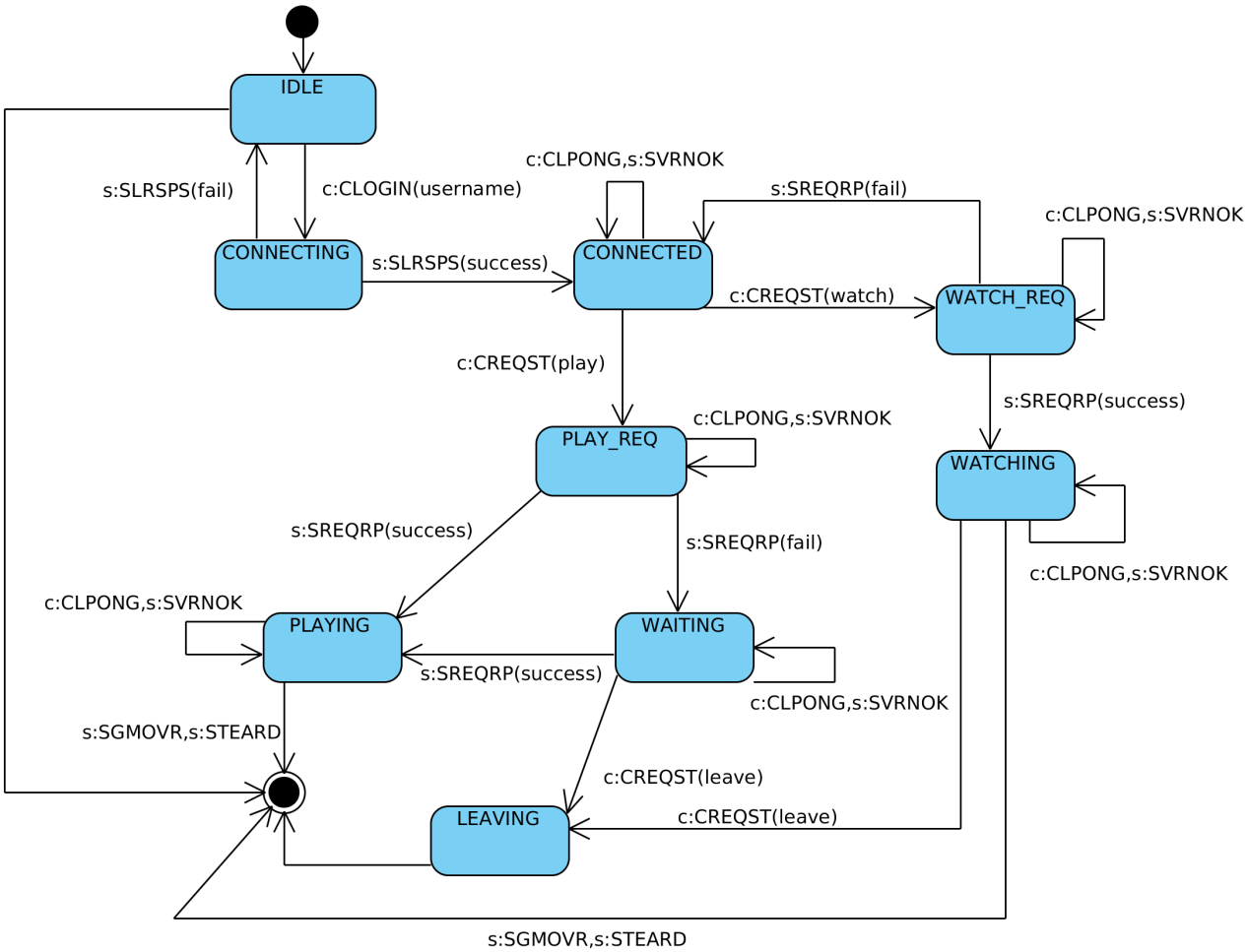
3.1. Client Object Model

Client class represents a client be it either a player or a watcher.



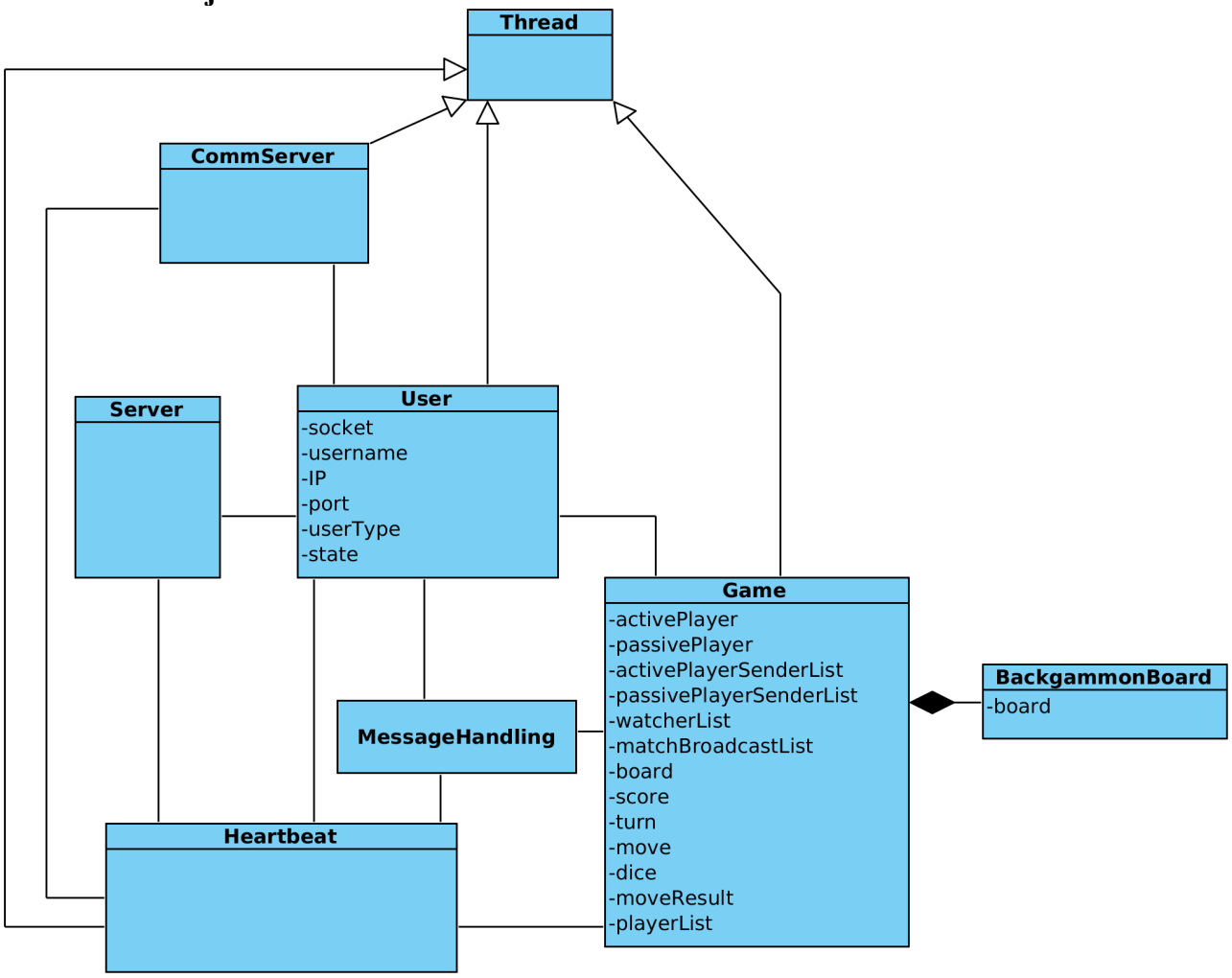
MessageHandling and BackgammonBoard are used both by client and server. Client creates a backgammon board only its request as either play or watch is accepted by the Server. Client has two endpoints to listen to after its login request is satisfied by the Server: (1) messages sent by the Server (2) input entered by the User. Python select.poll() or something similar will be used for this purpose.

2.2. Client State Machine



4. Server Architecture and Design

4.1. Server Object Model



Server is the main thread. Its sole purpose is to receive a connection request from a client, create a new User object (a new thread) and give its handling to it.

User handles 'play' and 'watch' requests in **connected** state and 'leave' request in **waiting** state. A unix domain socket is used between Heartbeat and User object. User connects to CommServer before entering into connected state. CommServer sends the client socket to Heartbeat. User listens to both client socket and unix domain socket it obtained from Comm server by using either select.poll() or select.select(). If CLPONG message is received from the client, it calls the ping method of Heartbeat object. If play request is satisfied, a Game object is created, all the information were passed to Game and User object wait() for an event by Game object. Python Event object will be used for this purpose.

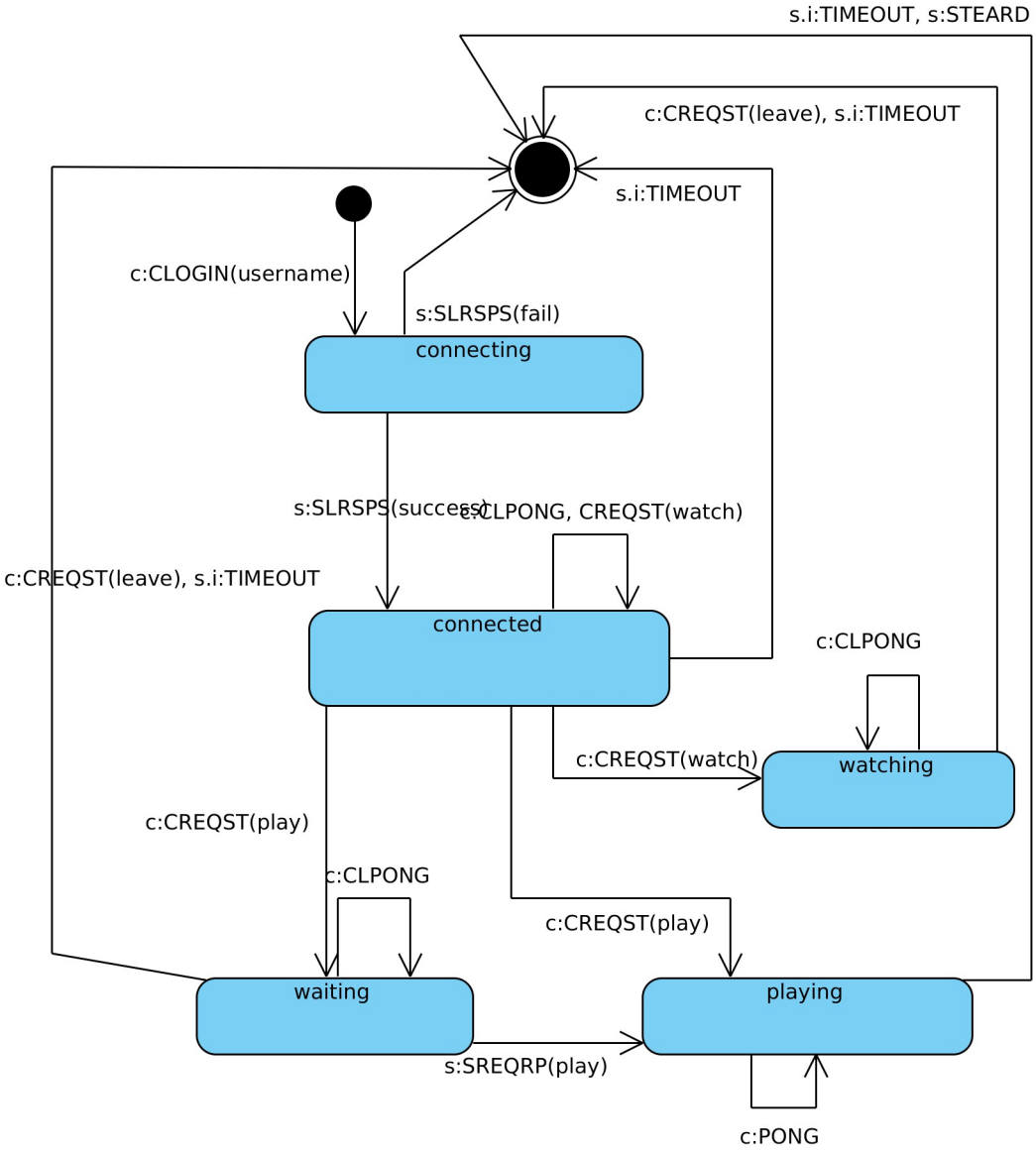
Heartbeat object runs in its own thread and started by Server. Every 30 seconds (heartbeat time), it sends all the clients SVPING message, which are registered to it. However, it does not collect response messages, instead User or Game thread collects the responses and informs Heartbeat. If Heartbeat object notices that a client is not responsive for more than 2 back-to-back PING, then it notifies User or Game thread, which are listening all the ports (a user connected to the server have 2 sockets: client socket and unix domain socket) by sending a message through the unix domain

socket.

CommServer is used to obtain a unix domain socket. Every user whose login is successful connects to CommServer to have a unix domain pair, which may going to be used by Heartbeat object to inform the user that his/her counterpart in the client side is dead.

Game also runs in its own thread. If a user wants to play a game and there is a user in the waiting room, then User object creates a Game object and passes all the information to it. Game object listens to player sockets (client socket + unix domain socket for Heartbeat) and watcher sockets. Actually "Playing State Machine" in the server side is controlled by Game object. User object handles most port of "Server State Machine".

4.2. Server State Machine



References

1. SWE544_RFC_1 document

2. SWE544_RFC_2 document

3. SWE544 lectures

4. Visual Paradigm, www.visual-paradigm.com