CT30A6000 Communications Software, Protocols and Architectures

Final project work: part one

Specifications Document

Group ID: 01
Members: Risto Laine, Nikolaos Paraschou
Topic: Freedom Game

Lappeenranta 19/03/2012

Table of Contents

1. Domain Specifics

Freedom (Cirovic & Sankovic 2012) is a is a two-player board game played on a square 10×10 board.

Equipment

Board 10×10, 50 white and 50 black pieces (stones) of the same value.

The goal of the game

To have more pieces, than your opponent, that live in a rows of 4 (in all possible directions), at the end of the game.

Play

1. Players alternately place pieces in the fields. Game begins by placing a white figure in any field on table, a player who is next in turn (black) is obliged to play his move to one of the empty fields that are adjacent field that was played the previous move (fields adjacent orthogonal or diagonal).
2. In a situation where a player can not play a move by the previous rule (all the adjacent fields are filled), he has the right ("freedom") to play anywhere on the empty field on table, and the other player continues to play by the rule 1. (in the adjacent field).
3. The game is over when all fields are filled. Once placed stones in the fields is not moving. The player who play last, has the right to choose to play or not to play (if it reduces his score).

End of the game

At the end of the game should be counted live figures. Live figure is figure who are in a row of 4 (orthogonal or diagonal). Arrays of larger or smaller length does not "give life" to figure. Score is expressed in the number of pieces that are live at the end of the game. The winner is the player with more points. If the numbers of points are equal, then it is the draw.

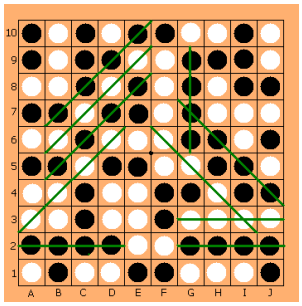


Illustration 1: example position on end of game, Black win: 23 – 15

Remark (example)

In the event when a figure is in more rows: in one direction is in a row of length 4, and in other directions rows with different lengths, the figure is still alive.

2. System Overview

The Freedom Game system follows the client-server communication model. Two basic communicating entities are identified in this model. One of the communication partners acts as client and one as server:

1. The server
   Implements the game logic, receives user's actions from the client and progresses the game, sends game updates to the clients.
2. The client
   Presents the freedom game GUI to the user, receives user input and forwards it to the server for processing, receives the server's updates and updates accordingly.

The server is able to support multiple game sessions, meaning that more than one pairs of players can play the game concurrently. The communication between the client and the server is made over UDP/IP.

The client is responsible to initiate a game session. The server must be running and be able to listen for incoming client connections. When a client connects, a new game session is initiated that runs on a separate thread. From this point on, it is this game session that will serve the client. When a second player connects to the same game session, the game can be started.

- The game server is responsible to constantly listen for incoming connection requests from the clients.
- The game session is responsible to “play” the game and update the two participating clients.

In this design we are following the publish-subscribe client-server model. The clients subscribe to the server (either by creating a new game or by joining one) and whenever there is new data the server pushes it to the subscribers (PUSH technology). For example, when player one makes a move the server receives the move, does some processing and pushes the move update to player two. Similarly, the server pushes player two’s move to player one.

The system overview is presented in detail in Illustration 2: Freedom Game system overview.

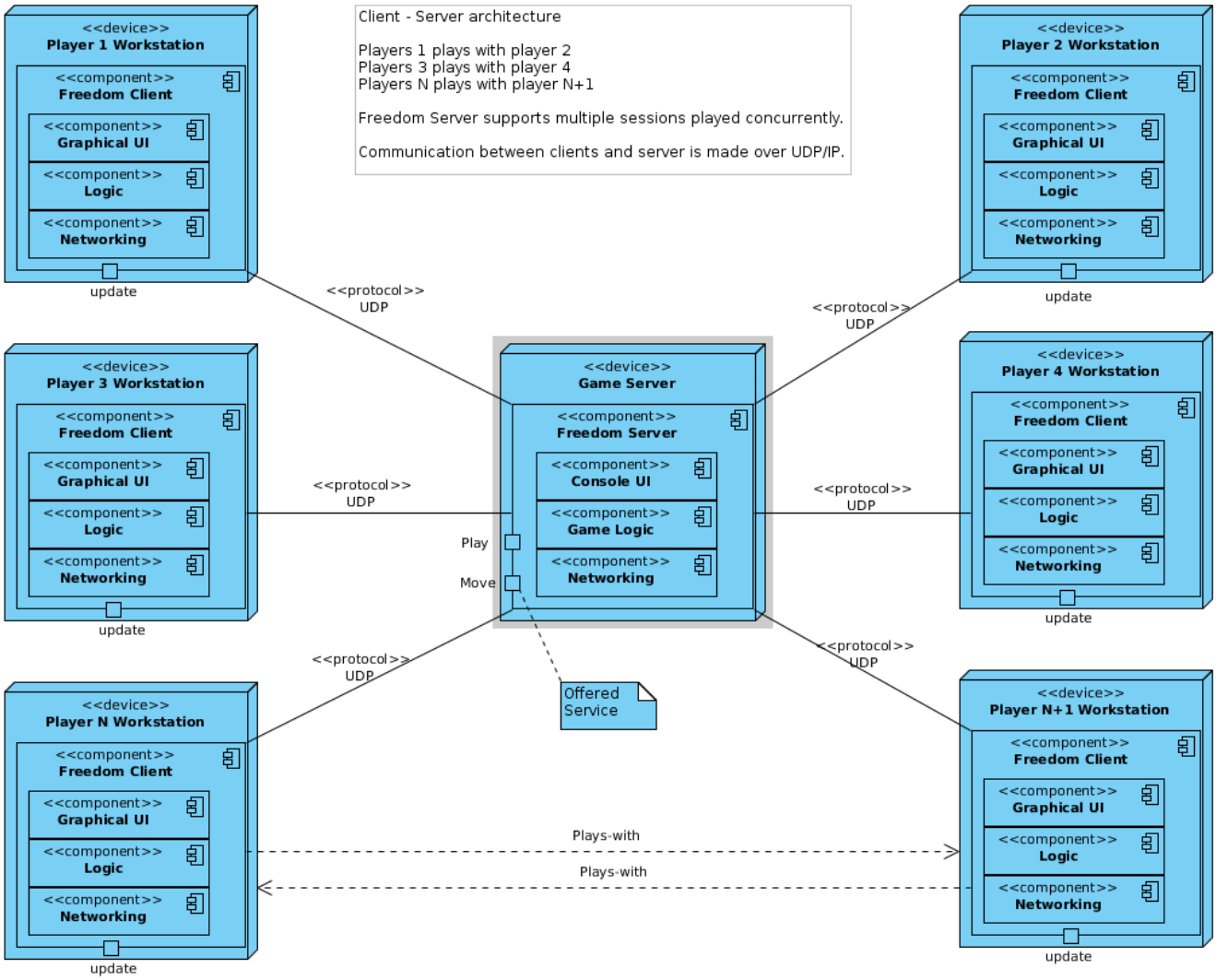


Illustration 2: Freedom Game system overview

3. Layer Service

3.1 Use Cases

On the highest level of abstraction, Freedom Game has three main use cases, Play, Move and Quit which are presented in the use case diagram of Illustration 3.

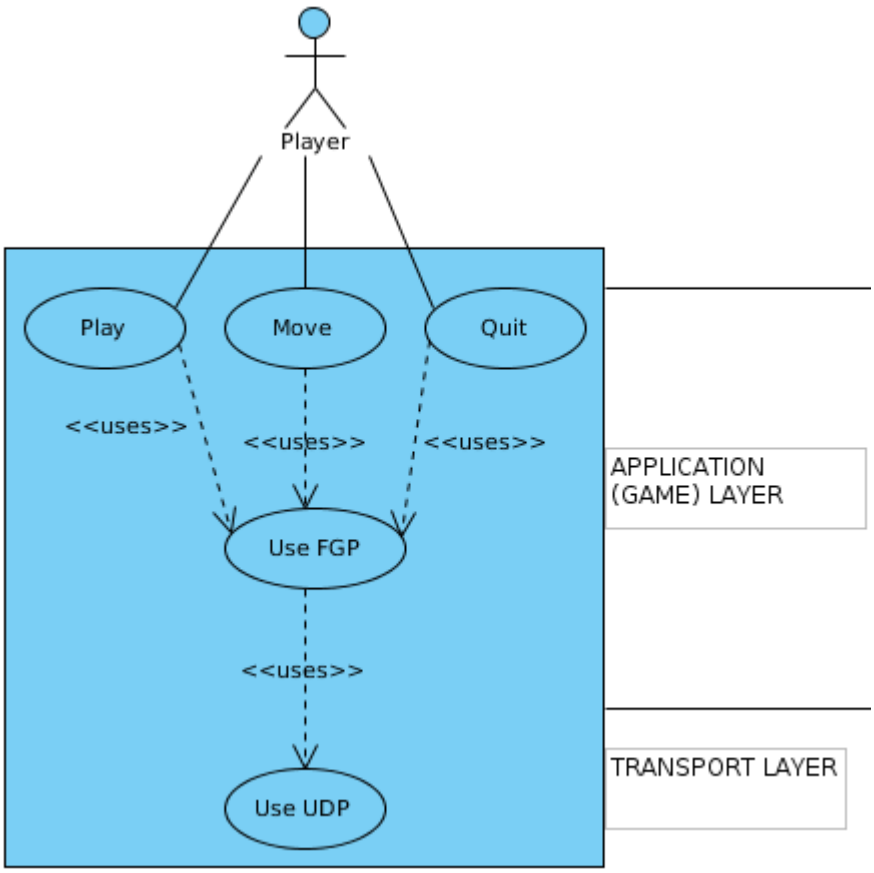


Illustration 3: Use Case Diagram

All these highest-level use cases make use of the use case Use FGP (Freedom Game Protocol) which uses the use case Use UDP (User Datagram Protocol).

The FGP receives the player's actions and depending on those, it sends the proper message to the server (play, move, drop). To do so, it utilizes the already existing UDP infrastructure.

The UDP provides data delivery service.

In the use case diagram of Illustration 4: FGP Use Cases, we present the requirements of the Freedom Game Protocol.

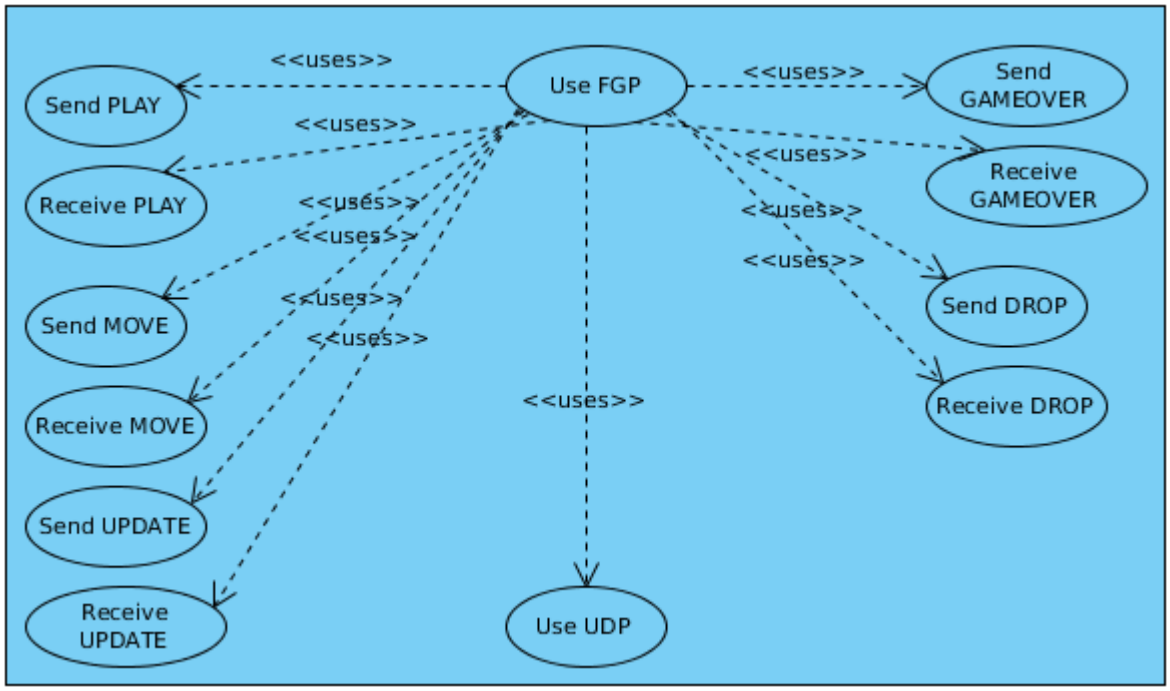


Illustration 4: FGP Use Cases

In the use case diagram of Illustration 5: Client-Server Use Cases, it is clearly visible what messages the client and the server can send and receive.

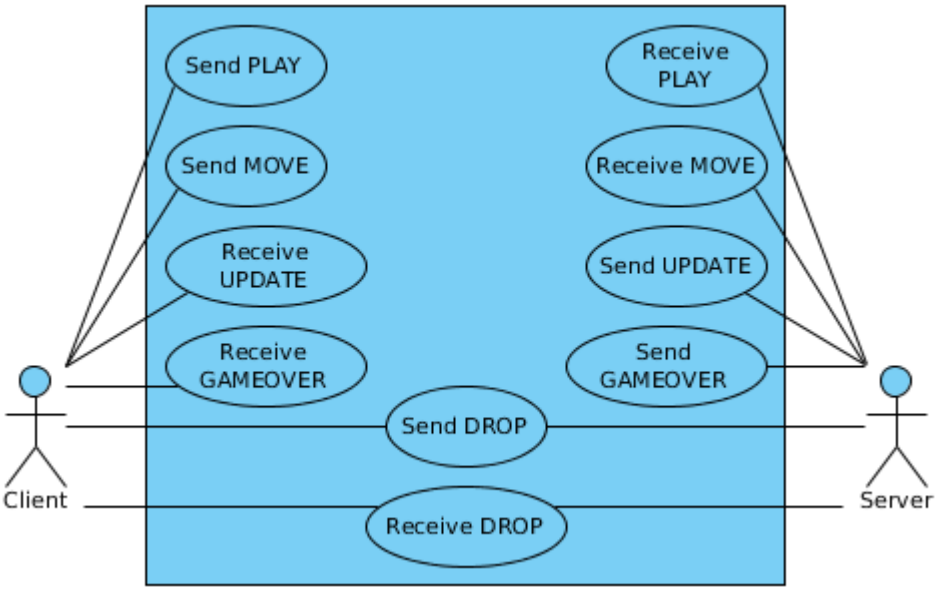


Illustration 5: Client-Server Use Cases

3.2 Service Descriptions

3.2.1 Game Layer

In this layer the game server and the game client are implemented. It contains the game logic, the data structures, the user interface etc. Since it is the upper most layer it provides only UI services. A User Interface component can use the services provided by the game layer to control the client. Similarly, some Control component can use the services provided by the game layer to control the server. The game layer uses the services provided by the transport layer (layer below) to establish communication between client and server.

FGP

FGP (Freedom Game Protocol) is implemented both on the client and the server parts of Freedom Game and it is responsible to send proper messages between the communicating entities depending on the user's actions. For example, when the user creates a game on the client side, the software of the client will notify FGP about the action and then FGP will forward the proper message to the server. Similarly, the server utilizes FGP to send messages to the clients. Basically, we are using FGP as an extra layer within the game layer. FGP is responsible to handle the communication between the clients and the server. We decoupled the communication logic from the game and placed it just below it. FGP utilizes UDP to perform the communication.

ID	Requirement statement
R01	Use UPD to transfer game messages.
R02	Send “play” message.
R03	Receive “play” message.
R04	Send “move” message.
R05	Receive “move” message.
R06	Send “update” message.
R07	Receive “update” message.
R08	Send “gameover” message.
R09	Receive “gameover” message.
R10	Send “drop” message.
R11	Receive “drop” message.

Table 1: FGP Service Requirements

ID	Constraint statement
C01	Use over the Internet (WAN) may be problematic due to the fact that the underlying layer (UDP) is unreliable.

Table 2: FGP Service Constraints

3.2.2 Transport Layer

The primary duty of the transport layer is to provide end-to-end communication from one application program to another (between game clients and the server). The transport layer provides communication services to the application layer and utilizes the internet layer to transmit data over the network.

UDP

The User Datagram Protocol or UDP provides the primary mechanism that Freedom Game uses to send and receive datagrams. UDP provides protocol ports used to distinguish among multiple game sessions executing on a single machine. That is, in addition to the data sent, each UDP message contains both a destination port number and a source port number, making it possible for the UDP software at the destination to deliver the message to the correct recipient and for the recipient to send a reply (Comer 2000).

UDP uses the underlying Internet Protocol to transport a message from one machine to another, and provides the same unreliable, connectionless datagram delivery semantics as IP. It does not use acknowledgements to make sure messages arrive, it does not order incoming messages, and it does not provide feedback to control the rate at which information flows between the machines. Thus, UDP messages can be lost, duplicated, or arrive out of order. Furthermore, packets can arrive faster than the recipient can process them (Comer 2000).

An application program that uses UDP accepts full responsibility for handling the problem of reliability, including message loss, duplication, delay, out-of-order delivery, and loss of connectivity.

ID	Requirement statement
R01	Transport data packet between sender and receiver.
R02	Pass received IP options to application layer.
R03	Application layer can specify IP options in Send.
R04	UDP passes IP options down to IP layer.
R05	Able to generate/check checksum.
R06	Bad IP source address silently discarded by UDP/IP.

Table 3: UDP Service Requirements

ID	Constraint statement
C01	Does not use acknowledgements to make sure messages arrive.
C02	Does not order incoming messages.
C03	Does not provide feedback to control the rate at which information flows between sender and receiver.

Table 4: UDP Service Constraints

4. Layer Model

The layer model in Illustration 6: Communication Architecture Diagram specifies the identified communicating entities within the game layer and the transport layer. It shows what are the entities of each layer and what are the communication channels between them. Additionally, SAP interfaces and peer PDU interfaces are illustrated along with the relationships between entities and interfaces.

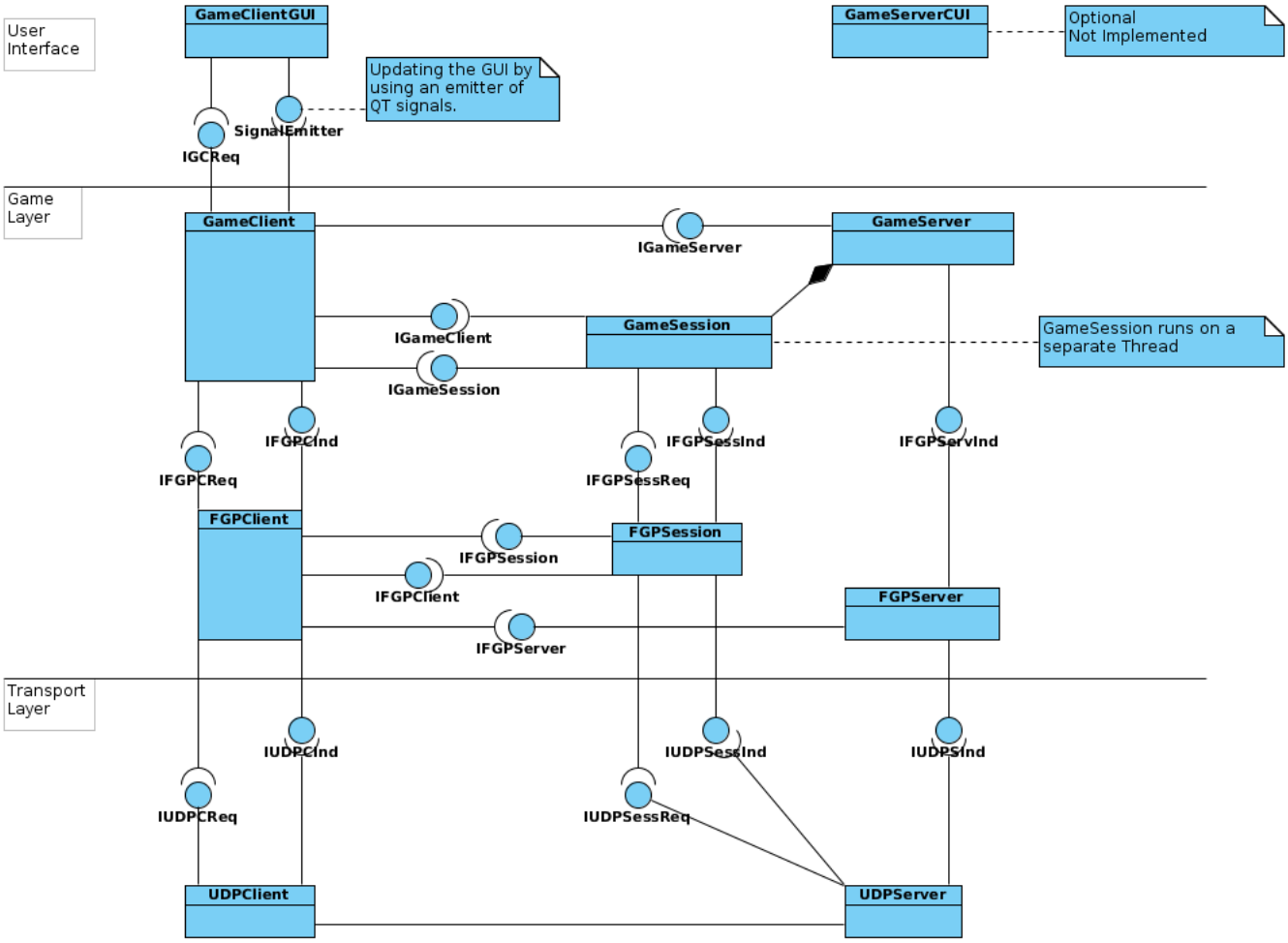


Illustration 6: Communication Architecture Diagram

5. State machine (FSM) Diagrams

5.1 GameServer FSM

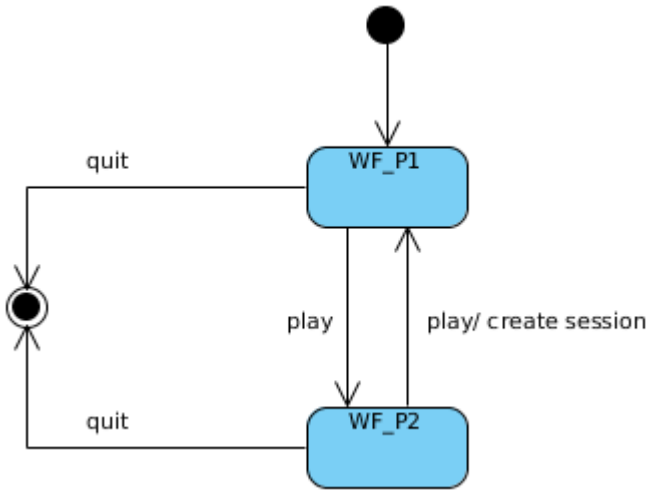


Illustration 7: Server state machine diagram

At initiation the server starts to listen for incoming play() messages from clients and more specifically from player one, WF_P1 state. Once a play() message is received from player one the server goes to WF_P2 state waiting for a play() message from player two. It is only after receiving a play() message from player two that the server will create a new session (on a separate thread) that will serve players one and two in playing the game. The server creates a game session for each pair of players requesting to play. For example, players one and two will play in session one, players three and four will play in session two, etc.

Msg/State	WF_P1	WF_P2
play	Goto WF_P2	Create new GameSession on a separate thread. Pass in the constructor the sendersqueue. Start the thread. Goto WF_P1
quit	Exit	Exit

Table 5: Game Server FSM table representation

5.2 GameSession FSM

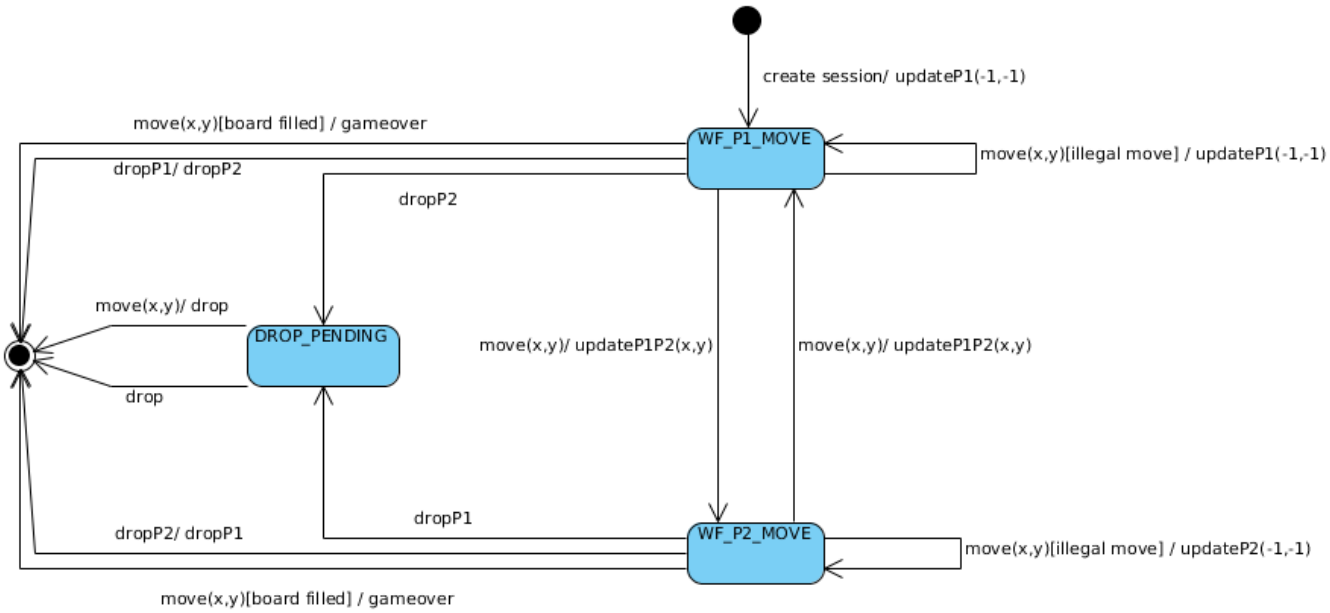


Illustration 8: Game session state machine diagram

When the game session is created it sends an update(-1, -1) message to player one indicating that it is his turn to play making the first move in the game.

Note: We use the update(...) message to update the clients. The update message can have three different meanings to the client depending on the parameters it carries:

1. update(-1, -1): Make your first move.
2. update(-1, -1): Illegal move play again.
3. update(x, y): Update your game board with opponent's (x,y) move.

In all three cases, the update message also carries the score of both players and a text message to the receiving client.

The initial state of the game session is WF_P1_MOVE waiting for player one's move. While in this state the game session can receive a move(x, y) message from player one which carries the coordinates of the move. There are three cases:

1. If the received move is illegal, the game session will send update(-1, -1) to player one indicating that his move was illegal requesting him to play again. There is no change of state.
2. If the move is legal and after the move the board is filled (game over), the game session will forward a gameover() message to both players and exit. The gameover() message carries the last move, in addition to the scores and a text message, so that the second player can update his game accordingly.
3. If the move is legal and after the move the board is not filled, the game session will forward the move to both players (with an update(x, y) message) and go to state WF_P2_MOVE waiting for player two's move. Player one receives the update as a confirmation that his move is accepted.

It also helps the client who made the last move update his score.

The same exactly cases apply in state WF_P2_MOVE.

While in state WF_P1_MOVE or WF_P2_MOVE, the game session can receive a drop message from any of the two clients meaning that the client who sent the drop has quit. In that case the game session has to react by notifying the other client that his opponent has quit. After that, the game session will terminate.

If the game session is in state WF_P1_MOVE and receives a drop message from player one, it will forward the drop to player two (who is listening) and terminate. If the game session is in WF_P1_MOVE and receives a drop message from player two, it will transition to state DROP_PENDING waiting for player one to make his move (or drop also). As soon as it receives player one's move, it will notify him that player two has quit and terminate.

Msg/State	WF_P1_MOVE	WF_P2_MOVE	DROP_PENDING

move	If (illegal move) send updateP1(-1,-1) to P1 and goto WF_P1_MOVE.
	If (board full) count score, send gameover(x,y) to P1 and P2 and exit.
	Else send update(x,y) to P1 and P2 and goto WF_P2_MOVE.
		If (illegal move) send updateP2(-1,-1) to P2 and goto WF_P2_MOVE.
		If (board full) count score, send gameover(x,y) to P1 and P2 and exit.
		Else send update(x,y) to P1 and P2 and goto WF_P1_MOVE.
			Reply with a DropPDU.
			Exit

drop	If dropP1 send dropP2 and exit. Else goto DROP_PENDING.	If dropP2 send dropP1 and exit. Else goto DROP_PENDING.	Exit

Table 6: Game Session FSM table representation

5.3 GameClient FSM

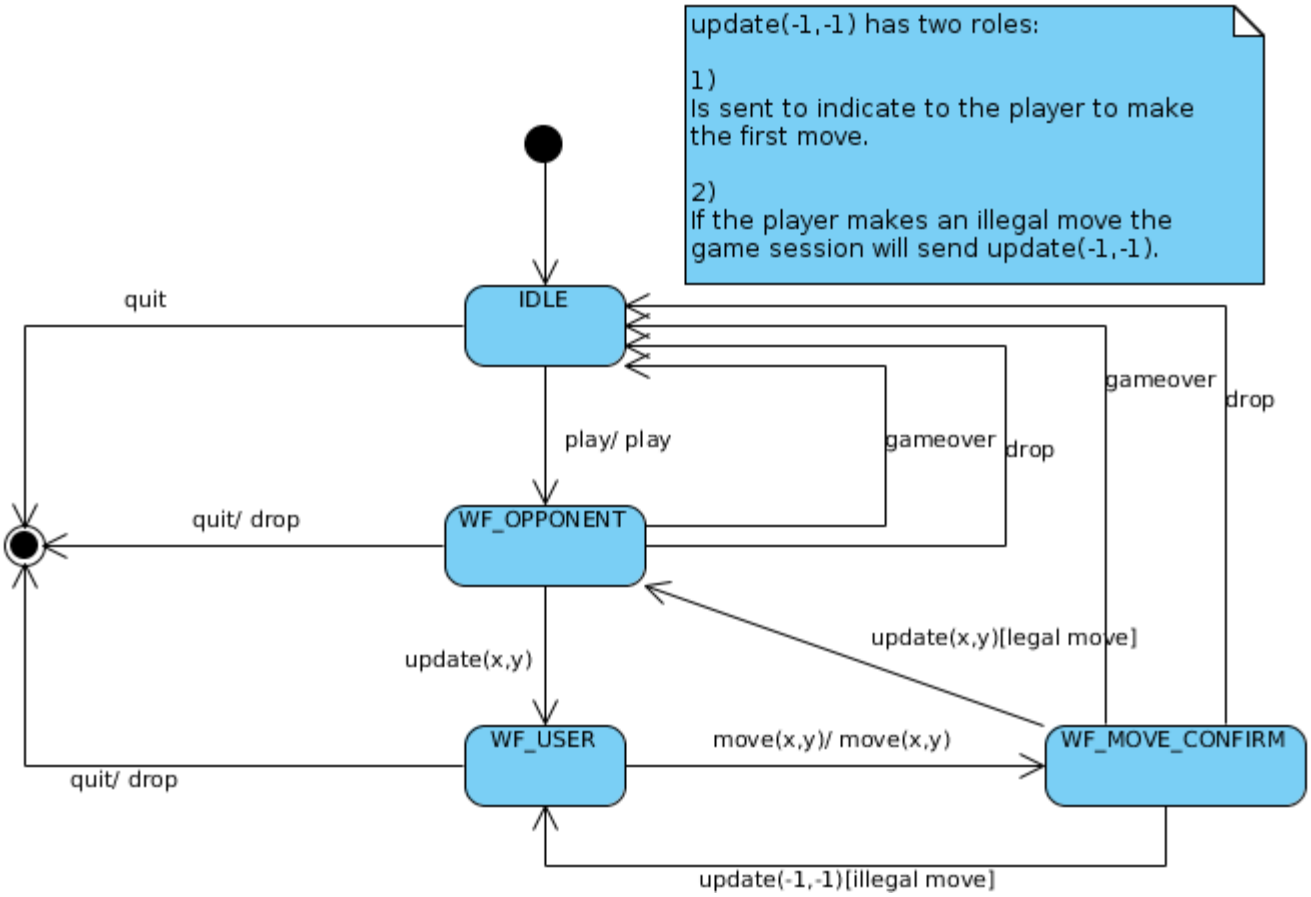


Illustration 9: Game client state machine diagram

The initial state of the client is IDLE. While in IDLE state, the client can receive a play() primitive from the player. In that case, the client will send a play() PDU to the server and go to WF_OPPONENT. In IDLE state the client can also receive a quit request in which case it quits.

In WF_OPPONENT state the client waits for an opponent to join (if not joined already) or play (if joined already). As soon as the opponent joins, the game session will tell player one to play with an update(-1, -1). If the opponent has already joined and the game is in progress, then the update will carry the opponent's move, in addition to the current score of both players and a text notification message. In that case, the game client will update the game and go to WF_USER, waiting for the user to make the next move.

If the opponent's move is the last one in the game (board is filled), then the game client will receive a gameover message (which carries the last move, the scores and a text message) and go to IDLE (game over).

If the opponent has quit the game, then the client will receive a drop message and goto IDLE.

In WF_USER state the client can receive the player's move. The client will send the move to the game

session and go to state WF_MOVE_CONFIRM waiting for a confirmation that the move is accepted. If the move was illegal, the client will receive an update(-1,-1) and go back to WF_USER. If the move was legal, the client will receive an update(x,y) (which carries also the current score for both players and a text message) and go to WF_OPPONENT.

In state WF_MOVE_CONFIRM the client can also receive a gameover() message, if the client has made the last move (this is a confirmation that the last move is accepted and that after the last move the game is over), or a drop message, if the opponent has quit. The gameover() message carries the last move's coordinates, the final score and a text message.

Finally, in states WF_OPPONENT and WF_USER the client can receive quit primitives from the user in which case it will send a drop message to the game session before exiting.

Msg/State	IDLE	WF_OPPONENT	WF_USER	WF_MOVE_CONFIRM
play	If (not first game) clear the board. Create PlayPDU. Set host and port on FGPClient using values given by user. Ask FGPClient to send PlayPDU. Goto WF_OPPONENT and ask FGPClient to listen for server reply.	Error	Error	Error
move	Error	Error	Get move's coordinates. Update the Game. Create MovePDU with coordinates. Ask FGPClient to send MovePDU. Goto WF_MOVE_CONFIRM and ask FGPClient to listen for server reply.	Error

update	Error	Get update from server.
		If (x=-1,y=-1) Set player color to white. Goto WF_USER.
		Else set player color to black if first move. Update the game.
		Goto WF_USER.
			Error	If (x=-1,y=-1) (illegal move) Update the Game (undo). Goto WF_USER.
				Else
				Goto WF_OPPONENT and ask FGPClient to listen for server reply.

gameover	Error	Get gameover text msg from server. If (board not full) Update the game. Goto IDLE.	Error	Get gameover text msg from server. Goto IDLE.
drop	Error	Get drop text msg from server. Goto IDLE.	Error	Get drop text msg from server. Undo last move on GUI. Goto IDLE.
quit	Exit	Create DropPDU. Ask FGPClient to send DropPDU. Exit.	Create DropPDU. Ask FGPClient to send DropPDU. Exit.	Error

Table 7: Game Client FSM table representation

6. Abstract Message Definitions

We have already presented in Illustration 6: Communication Architecture Diagram all the interfaces used in the system (both for primitives and PDUs). Below, we will discuss these interfaces and their messages in more detail.

6.1 Primitives

In this section we describe the primitives used by the communication system.

6.1.1 GameClient Primitives

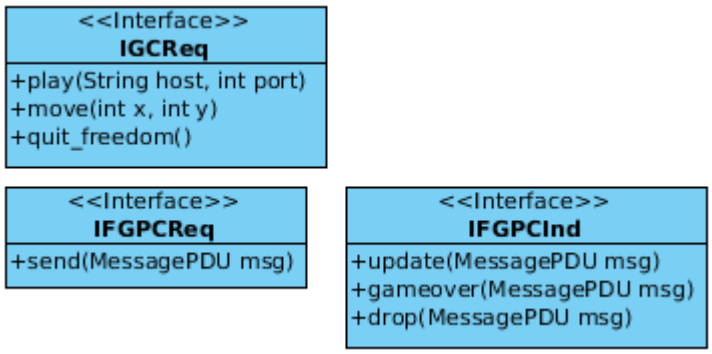


Illustration 10: Client Primitives

The interface IGCReq is used by the user through the GameClientGUI to send requests to the GameClient. The primitives offered are play(String host, int port), move(int x, int y) and quit_freedom(). With the play(String host, int port) primitive we request the GameClient to send a play() PDU to the GameServer. With the move(int x, int y) primitive we request the GameClient to send a move() PDU containing our move to the GameSession. With the quit_freedom() primitive we request the client to send a drop() PDU to the GameSession.

The GameClient is using the IFGPCReq to forward the PDUs to the server by using the send() primitive. The parameter provided is the PDU that will travel to the server or session.

The GameClient can be notified ("indicated") about updates coming from the GameSession through the IFGPCInd interface which provides the primitives update(), gameover() and drop().

The GameClient can update the GameClientGUI by emitting Qt Signals through a custom signal emitter (implementation detail due to implementation restrictions by PyQt (signals)).

6.1.2 GameSession Primitives

Illustration 11: Session Primitives

The GameSession uses the IFGPSessReq to forward PDUs to the clients through the send() primitive. The parameters provided are the PDU that will travel to the player and the player.

The interface IFGPSessInd is provided by the GameSession to receive move() and drop() indications from the FGPSession coming from the players.

6.1.3 GameServer Primitives

Illustration 12:
Server Primitives

The GameServer is capable only to listen for play() requests from the players. So, there is only an indication interface, IFGPServInd, which indicates the GameServer about play() requests from the clients. The GameServer does not send any messages to the clients.

6.2 PDUs

In the following illustration we present the abstract message definitions.

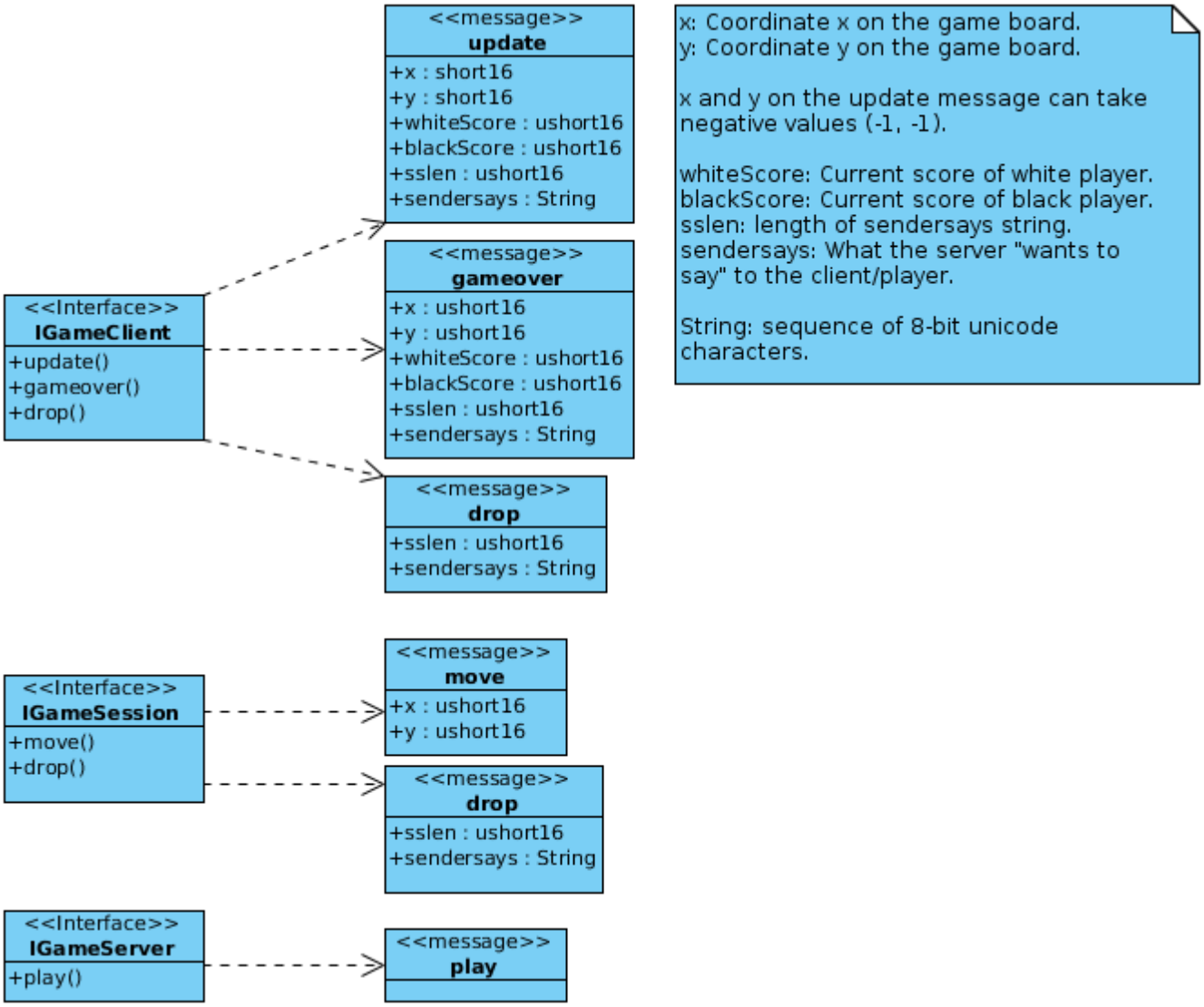


Illustration 13: Abstract Message Definitions

6.2.1 GameClient PDUs

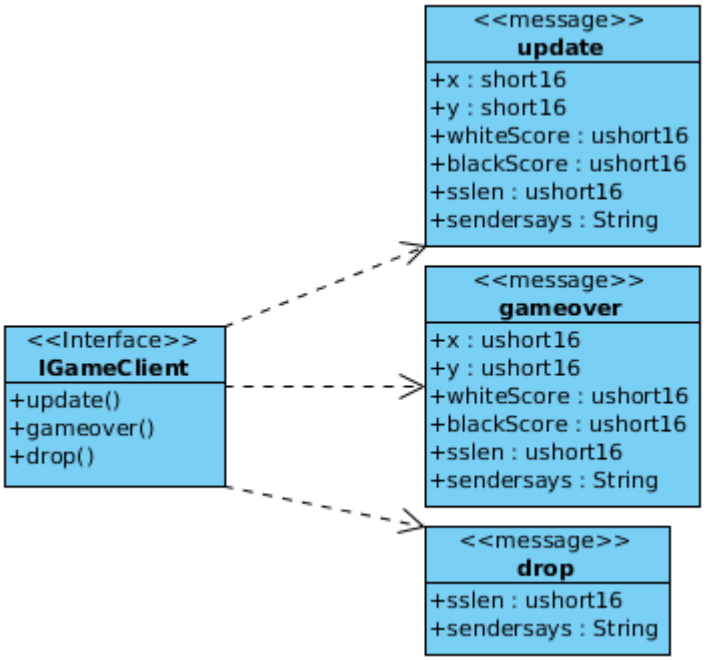


Illustration 14: IGameClient PDUs

The GameClient can receive from the GameSession the PDUs update(), gameover() and drop().

The update() PDU is meant to update the client according to the opponent's move or let the client know that the game has just started and it is his turn (x = -1 and y = -1) or indicate an illegal move (x = -1 and y = -1). In all three cases the PDU carries two numbers, x and y which are of type short16 (can take negative values). Those numbers represent the move's coordinates on the board. Additionally, the update() PDU carries the score of both players and a text message indicating what the session wants to tell to the client/player.

The gameover() PDU notifies the client that the game is over and lets the client know what was the last move. The gameover() PDU is very similar to the update() PDU with one difference. The coordinates x and y are of type ushort16 because they don't take negative values (only positive coordinate values).

The drop() PDU indicates to the client that his opponent has quit. It carries a text notification message indicating what the session wants to tell to the player.

6.2.2 GameSessionPDUs

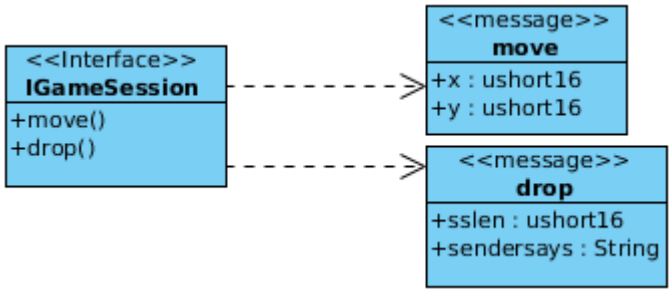


Illustration 15: IGameSession PDUs

The GameSession can receive from the GameClient the PDUs move() and drop().

The move() PDU carries the player's move (x and y are the move's coordinates on the game board). The parameters x and y are of type ushort16 (16-bit unsigned short).

The drop() PDU indicates to the session that the client has quit. The text notification message carried by the drop PDU is not used in this case (from client to session).

6.2.3 GameServerPDUs

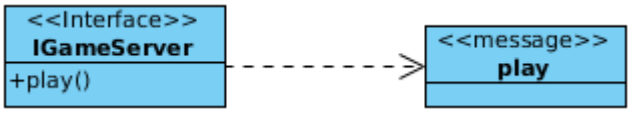


Illustration 16: IGameServer PDUs

The GameServer can receive from the GameClient the PDU play().

The play() PDU carries no information. It just lets the GameServer know that the GameClient wants to play. The GameServer, depending on its state, will either start a new GameSession and register the player to play in it or wait for a second play PDU from another player before starting a new GameSession.

7. Concrete Message definitions

The following section contains the byte count coding of the messages provided by the interfaces.

7.1 Message boundaries

Every message has a common header defining the type of the message and size of the message.

IGameClient

Message update

	0..7	8..15	16..23	24..31
0	type		size	
1	x		y	
2	whitescore		blackscore	
3	strsize		str...	
...	...str		PADDING	

Message gameover

	0..7	8..15	16..23	24..31
0	type		size	
1	x		y	
2	whitescore		blackscore	
3	strsize		str...	
...	...str		PADDING	

Message drop

	0..7	8..15	16..23	24..31
0	type		size	
1	strsize		str...	
...	...str		PADDING	

IGameSession

Message move

	0..7	8..15	16..23	24..31
0	type		size	
1	x		y	

Message drop

	0..7	8..15	16..23	24..31
0	type		size	
1	strsize		str...	
...	...str		PADDING	

IGameServer
Message play

	0..7	8..15	16..23	24..31
0	type		size	

7.2 Type definition catalog

- type: enumeration as an unsigned short 16-bit
- size: unsigned short 16-bit
- x in update: short 16-bit (can take negative values)
- y in update: short 16-bit (can take negative values)
- x in move: unsigned short 16-bit
- y in move: unsigned short 16-bit
- whitescore: unsigned short 16-bit
- blackscore: unsigned short 16-bit
- strsize: unsigned short 16-bit, number of 16-bit Unicode characters in a string

Message	Message type id
PlayPDU	1
MovePDU	2
UpdatePDU	3
GameOverPDU	4
DropPDU	5

Table 8: Type enumerations definition

7.3 List of Used Types (Type-catalog)

- unsigned short 16 bit
- short 16 bit
- string

REFERENCES

Comer, E. D. 2000. Internetworking with TCP/IP Vol I: Principles, Protocols, and Architecture. Fourth Edition. New Jersey. Prentice Hall

Cirovic, V., Sankovic, N. 16.03.2012, FREEDOM New abstract strategy game, http://freedomtable.wordpress.com/

APPENDIX

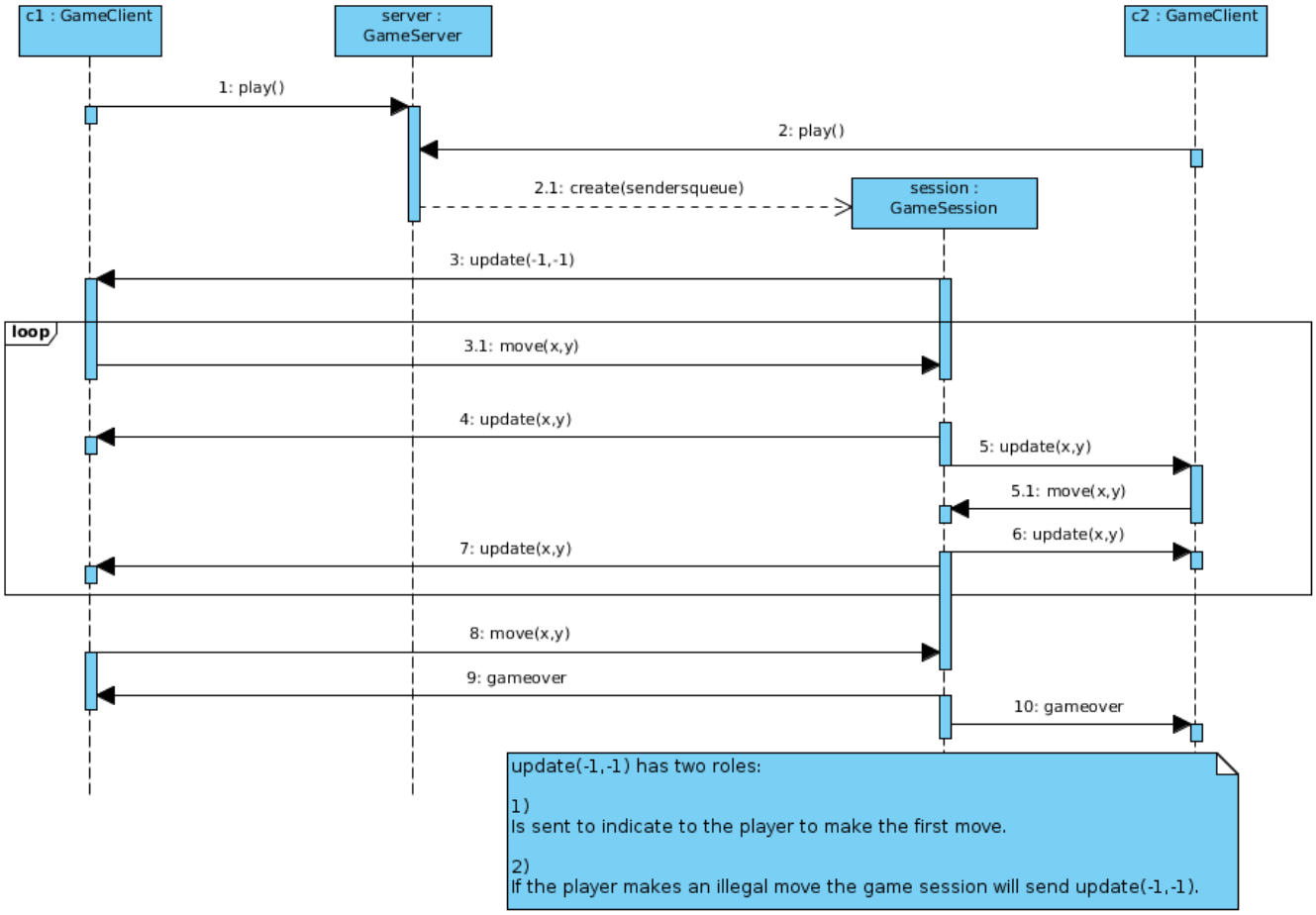


Illustration 17: Create Join Play Gameover sequence diagram

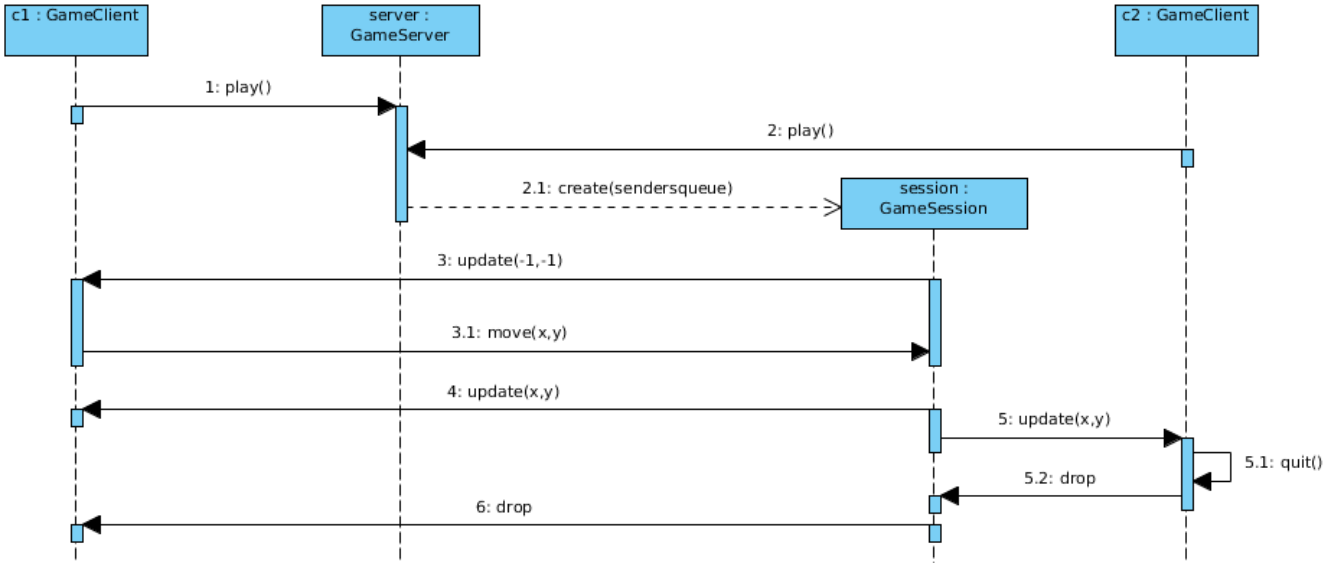
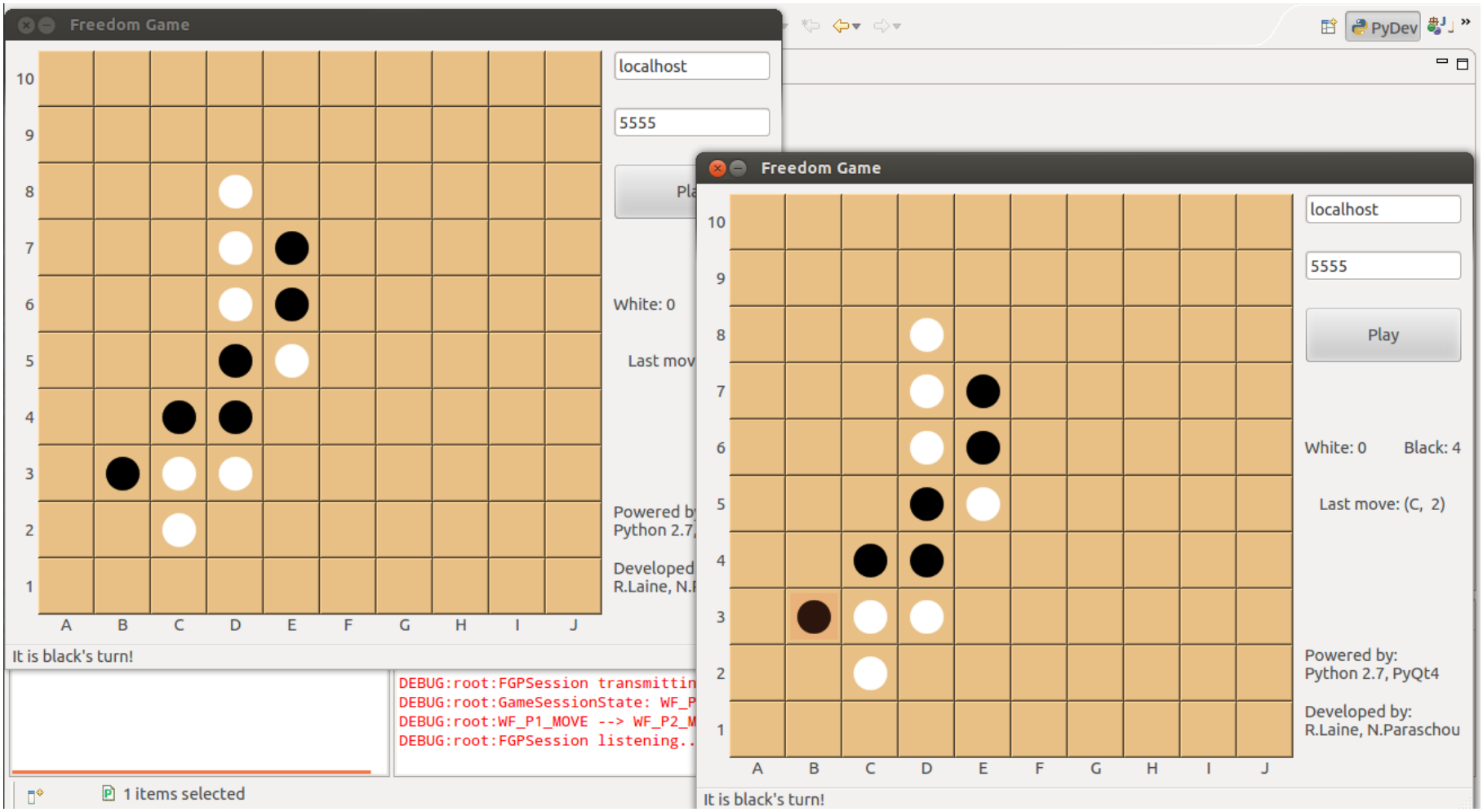


Illustration 18: Drop sequence diagram

Freedom.png


Freedom Game
localhost
5555
Play
White: 0
Black: 4
Last move: (C, 2)
Powered by:
Python 2.7, PyQt4
Developed by:
R.Laine, N.Paraschou
It is black's turn!
PyDev
DEBUG:root:FGPSession transmittin
DEBUG:root:GameSessionState: WF_P
DEBUG:root:WF_P1_MOVE --> WF_P2_M
DEBUG:root:FGPSession listening..
1 items selected