UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Department of Electrical, Electronic and Computer Engineering

EPE 321- Communication Protocol

Pieter Kok

Last updated: Monday 14th September, 2020

Table of Contents

Acknowledgements

This communication protocol was adapted from the protocol originally compiled by Mohamed Ameen Omar and the EPE 321 class of 2018.

1 Change Log

1.1 Version 1.0 (14/09/2020)

- Initial Release

2 Introduction

This document seeks to standardize the communication protocol being used in the design, development, and implementation of the Rubber Contract Bridge game being developed by the EPE 321 students at the University of Pretoria. This document will serve as a standard for all messages that should be sent or received by a client and host programs. This only serves as a guide of the minimum protocol that is expected, however groups may add to or alter this protocol as they see fit.

3 Constraints

The following are the constraints that should be adhered to for all communication between the entities in this project:

- All network communication is done using Web-sockets.
- Entities within the game are identified by their local network IP address and port.
- A client connects to the host via the host's local network IP Address and port.
- No client may directly communicate with another client.
- Network communication need not support remote networks, therefore the host and all the clients must reside within the same local area network.
- The format used for messages will be in the JSON format.
- All JSON files sent should include a ***"Typ"*** field.
- If a field within the JSON file is not supported by the host or client, this field must simply be ignored.
- **All** messages sent should be within a JSON object. The data should not be sent solely within a JSON array.

3.1 Bridge Rules

The rules specified in `https://www.bicyclecards.com/how-to-play/bridge/` should be adhered to.

4 Web-socket Secure

The connection between client and server application is a Secure Web-socket (`wss://`) on the application level of the OSI model. The server application (host) hosts a Secure Web-socket Server, which the client application connects to. The Secure Web-socket can be hosted on any port.

4.1 Qt implementation

A suggestion for the implementation is to use the `QWebsocket`, and the `QWebSocketServer` class provided by Qt. Messages should be sent as text, i.e. using the `sendTextMessage` function of the QWebSocket class.

4.1.1 Server - Web-socket Secure Example

An example for the server implementation can be found at `http://doc.qt.io/qt-5/qtwebsockets-sslechoserver-example.html`.

4.1.2 Client - Web-socket Secure Example

An example for the client implementation can be found at `http://doc.qt.io/qt-5/qtwebsockets-sslechoclient-example.html`.

4.2 Encryption

Set `QSsl::TlsV1SslV3` as the SSL protocol. The server will specify the encryption - the client will use the server's SSL protocol, determined automatically during setup of the Web-socket. Since most or all server SSL certificates will be self-signed, this must be tolerated by the server and client. To get started on accepting self-signed certificates, look at `http://doc.qt.io/qt-5/qwebsocket.html#ignoreSslErrors-1`.

4.2.1 Certificate Generation

The host (but not the client) has to supply an SSL certificate for the Web-socket Secure Server. This can be generated through the following command:

```
1 openssl req -x509 -nodes -newkey rsa:4096 -keyout localhost.key -out localhost.
    cert -days 24855
```

Each server should have its own unique certificate. If someone else gets access to the certificate files, the encryption is no longer secure.

5 Message Structure

All messages sent must in a JSON file. JSON stands for JavaScript Object Notation and allows for data to be sent from host to client and from client to host seamlessly. JSON objects are made up of a series of "key-value" pairs. The keys within a JSON object may be any string, while the values for each associated key may be a String, Number, Boolean, null, array or another object. JSON objects are represented within curly braces ({ }), arrays are represented within square brackets ([ ]), a key is separated by it's value with a colon (:) and all key-value pairs are separated by commas. The examples below further clarify the syntax and structure of JSON objects.

5.1 JSON message examples

5.1.1 A simple JSON object

```
1 { "name": "John", "age": 31, "city": "New York" }
```

Notice that the JSON object begins with '{' and the object is terminated with '}'. Each key-value pair is separated by a comma, and every *key* is a String.

5.2 JSON objects with arrays

```
1  { "Books" : [
2              { "Name"  : "Software Engineering", "price" : 700 },
3              { "Name"  : "Guide to JSON", "price" : 400 }
4              ],
5
6              "Modules"  : [
7                 { "Name"  : "EPE321", "fees" : 8000 },
8                 { "Name"  : "EAI320", "fees" : 7500 }
9                 ]
10 }
```

6 Disconnections

A player may be disconnected due to a poor connection to the host, inadequate response time, failure to respond to a connection verification request made by the host or by the client itself. In the event that a client loses connection to the host due to the network being too slow or failure to send a response for a request received, the host will detect this and should end the game and disconnect all other clients.

7 Messages

Type	**Description**	**Origin and destination**
CONNECT_REQUEST	Message to request a connection to participate in the game.	Client to host.
CONNECT_UNSUCCESSFUL	Message to indicate that the connection was rejected.	Host to client.
CONNECT_SUCCESSFUL	Message to indicate that the connection was successful.	Host to client.
LOBBY_UPDATE	Message to update clients of a new player joining the game.	Host to all.
BID_START	Message to indicate the start of the bidding phase.	Host to all.
BID_REQUEST	Message to request a bid from a player.	Host to client
BID_SEND	Message to indicate the bid chosen by a player.	Client to host.
BID_UPDATE	Message to indicate the valid bid that was just made.	Host to all.
BID_END	Message to indicate the end of the bidding stage.	Host to all.
PLAY_START	Message to indicate the start of the playing phase.	Host to all.
MOVE_REQUEST	Message to request a move from a player.	Host to client
MOVE_SEND	Message to send the move to be made.	Client to host.
MOVE_UPDATE	Message to indicate the valid move that was just made.	Host to all.
TRICK_END	Message to indicate the end of a trick.	Host to all.

PLAY_END	Message to indicate the end of the play stage.	Host to all.
SCORE	Message to indicate the updated score of the game.	Host to all.
PING	Message to verify if a client/host is still connected.	Client to host, host to client.
PONG	Message to respond to a PING.	Client to host, host to client.
DISCONNECT_PLAYER	Message to indicate that a player has disconnected or has been disconnected by the host.	Host to all.

8 JSON Messages

This section provides the key-value pairs and the structure of each message sent over the TCP/IP Connection. Every JSON object should have two mandatory *keys*, the **Type** and **Id** keys. The **Type** key must have the specific standardized message type as its value and the **Id** key is a unique identifier for each message, generated by the entity generating and sending the message. All **Id** keys should have a unique generated integer as its value.

8.1 CONNECT_REQUEST

```
{
    "Type": "CONNECT_REQUEST",
    "Id": 2,
    "Password": "password123",
    "Alias": "Player123",
}
```

The ***Password*** can be any value if the server has not set a password, else it must match the server password. The password is only used to accept/reject connection requests by client. The alias key may have any value that the client would like to be publicly known as during the duration of the game.

8.2 CONNECT_UNSUCCESSFUL

```
{
    "Type": "CONNECT_UNSUCCESSFUL",
    "Id": 3,
    "Description": "LOBBY_FULL"
}
```

The ***Description*** key may take on the values: *LOBBY_FULL* or *OTHER*.

8.3 CONNECT_SUCCESSFUL

```
{
    "Type": "CONNECT_SUCCESSFUL",
    "Id": 4
}
```

8.4 LOBBY_UPDATE

```
{
    "Type": "LOBBY_UPDATE",
    "Id": 6,
    "PlayerPositions": [
    { "Position": "N", "Alias": "P1" },
    { "Position": "E", "Alias": "P2" },
    { "Position": "W", "Alias": "P3" }
    ]
}
```

The ***PlayerPosition*** key consists of an array of objects that each consist of keys ***Position*** and ***Alias*** that denote the alias of the player in each position. Only positions that have been filled should be listed.

8.5 BID_START

```
{
    "Type": "BID_START",
    "Id": 15,
    "Cards": [
                {"Suit" : "H", "Rank" : ["2", "7"] },
                {"Suit" : "S", "Rank" : ["5", "A"] },
                {"Suit" : "C", "Rank" : ["6", "J", "Q", "K"] },
                {"Suit" : "D", "Rank" : ["2", "9", "10", "A"] }
            ],
    "Dealer": "N"
}
```

The ***Cards*** key holds an array of 13 JSON objects. Each element holds the **Suit** and the **Rank** for each suit. This array will represent the cards dealt to a player and will be specifically made for each client.

The Suit holds any of the following values:

- **"H"** for the Hearts suit.
- **"S"** for the Spades suit.
- **"C"** for the Clubs suit.
- **"D"** for the Diamonds suit.

The Rank holds any of the following values:

- Any number from 2-10.
- **"A"** for the Ace rank.
- **"K"** for the King rank.
- **"Q"** for the Queen rank.
- **"J"** for the Jack rank.

The ***Dealer*** key will signify the position of the player who is the dealer for the current game and as such will make the first bid during the bidding phase of the current game.

8.6 BID_REQUEST

```
{
    "Type": "BID_REQUEST",
    "Id": 18
}
```

8.7 BID_SEND

```
{
    "Type": "BID_SEND",
    "Id": 22,
    "Bid": {"Suit" : "H", "Rank" : "5" }
}
```

The ***Bid*** key holds a JSON object of the bid made by a player. Each element holds the **Suit** and the **Rank** for each suit. This array will represent the valid bids that may be made by a player based off of the player's cards and the previous bids made.

The Suit holds any of the following values:

- **"H"** for the Hearts suit.
- **"S"** for the Spades suit.
- **"C"** for the Clubs suit.

- **"D"** for the Diamonds suit.
- **"DOUBLE"** for a DOUBLE.
- **"REDOUBLE"** for a REDOUBLE.
- **"PASS"** for a PASS.
- **"NT"** for a NO TRUMP.

The Rank holds any of the following values:

- Any number from 1-7.
- a ***null*** key when appropriate.

8.8 BID_UPDATE

```
{
    "Type": "BID_UPDATE",
    "Id": 29,
    "Bid": {"Suit" : "H", "Rank" : "5" },
    "Player": "N"
}
```

The ***Bid*** key holds a JSON object of the bid made by a player. The ***Player*** key holds the position of the player who made the bid as it's value.

8.9 BID_END

```
{
    "Type": "BID_END",
    "Id": 31,
    "Contract": {"Suit" : "H", "Rank" : "3", "IsDouble" : True, "
   IsRedouble" : False },
    "Trump": "H",
    "Declarer": "N",
    "Dummy": "S"
}
```

The ***Contract*** key holds a JSON object that represents the Contract for the current game. The ***Trump*** key holds the Trump suit or a null value (in the event of a NT). The **Declarer** and ***Dummy*** keys holds the position of the Declarer and the Dummy, respectively, for the current game. The ***IsDouble*** and ***IsRedouble*** are boolean values indicating if the bid has been doubled or redoubled.

8.10 PLAY_START

```
{
    "Type": "PLAY_START",
    "Id": 32,
    "DummyCards": [
                {"Suit" : "H", "Rank" : ["2", "7"] },
                {"Suit" : "S", "Rank" : ["5", "A"] },
                {"Suit" : "C", "Rank" : ["4", "6", "J", "Q"] },
                {"Suit" : "D", "Rank" : ["2", "9", "10", "A"] }
                  ]
}
```

The ***DummyCards*** key holds an array of up to 13 JSON objects which represent the cards held by the Dummy player for the current game (Only 4 are shown to save space and eliminate redundant representations).

8.11 MOVE_REQUEST

```
{
    "Type": "MOVE_REQUEST",
    "Id": 37,
    "MoveDummy": False
}
```

The ***MoveDummy*** key holds a Boolean value indicating if the move is for the declaring player to play for the dummy.

8.12 MOVE_SEND

```
{
    "Type": "MOVE_SEND",
    "Id": 39,
    "Move": {"Suit" : "H", "Rank" : "3" }
}
```

The ***Move*** key holds the card played by a player as its key represented as a JSON object.

8.13 MOVE_UPDATE

```
{
    "Type": "MOVE_UPDATE",
    "Id": 43,
    "Move": {"Suit" : "H", "Rank" : "5" },
    "Player": "N"
}
```

The ***Move*** key holds a JSON object of the move made by a player. The ***Player*** key holds the position of the player who made the move as it's value.

8.14 TRICK_END

```
{
    "Type": "TRICK_END",
    "Id": 45,
    "WinningPartnership": "NS"
}
```

The ***WinningPartnership*** key holds a value of "NS" or "EW" indicating which partnership won the current trick.

8.15 PLAY_END

```
{
    "Type": "PLAY_END",
    "Id": 49,
    "WinningPartnership": "NS"
}
```

The ***WinningPartnership*** key holds the value of the partnership who won the current iteration of the *play phase*. The ***TricksWon*** key will be an integer value from 1-13 indicating the number of tricks won by the winning partnership.

8.16 SCORE

```
{
    "Type": "SCORE",
    "Id": 60,
    "NSscores": {
            "overtricks" : 100,
```

```
            "undertricks" : 0,
            "honors" : 0,
            "vulnerable" : 0,
            "double" : 0,
            "redouble" : 0,
            "slam" : 0,
            "unfinished" : 0,
            "trickScore" : 50
    },
    "EWscores": {
            "overtricks" : 0,
            "undertricks" : 0,
            "honors" : 0,
            "vulnerable" : 0,
            "double" : 0,
            "redouble" : 0,
            "slam" : 0,
            "unfinished" : 0,
            "trickScore" : 0
    }

}
```

The ***NSscores*** and ***EWscores*** keys will each hold a JSON object. Within each object a breakdown of the types of scores will be held. The score sent is for a game and will be sent at completion of each game.

8.17 GAME_END

```
{
    "Type": "GAME_END",
    "Id": 100,
    "WinningPartnership" : "NS"
}
```

The ***WinningPartnership*** key holds the value of the partnership who won the current game.

8.18 PING

```
{
    "Type": "PING",
```

```
    "Id": 104
}
```

8.19 PONG

```
{
    "Type": "PONG",
    "Id": 107
}
```

8.20 DISCONNECT_PLAYER

```
{
    "Type": "DISCONNECT_PLAYER",
    "Id": 109,
    "PlayerPos": "N"
}
```

The ***PlayerPos*** key holds the position of the player who has been disconnected from the current game session and as such will be replaced by a new AI player. The game will continue with minimal interruption.

9 Glossary

Term	Description
Move	Describes the act of a player placing his card on the table.
Trick	Consists of four moves.
Game	Consists of a set of 13 tricks.
Rubber	Consists of a number of games.