24 March 2024

INTERNET TECHNOLOGY

PROTOCOL SPECIFICATION FOR SERVER

Edited by Badr Belarbi (508343) & Brice Nana Nyamgam (490377)

Class: DHI2V.So

Supervised by Gerralt Gottermaker

Contents

Level 1

The protocol is described below. The protocol describes the following scenarios:

- Setting up a connection between client and server.
- Broadcasting a message to all connected clients.
- Periodically sending heartbeats to connected clients.
- Disconnection from the server.
- Handling invalid messages.

In the description below, C -> S represents a message from the client C is sent to server S. When applicable, C is extended with a number to indicate a specific client, e.g., C1, C2, etc. The keyword "others" indicates all other clients except those who made the request.
Messages can contain a JSON body. Text between "<" and ">" are placeholders.

1. Establishing a connection

The client first sets up a socket connection to which the server responds with a welcome message. The client supplies a username on which the server responds with an OK if the username is accepted or an ERROR with a number in case of an error.
Note: A username may only consist of characters, numbers, and underscores ('_') and has a length between 3 to 14 characters.

Happy flow:

Client sets up the connection with the server.

```
S -> C: WELCOME {“msg”: “< Welcome to the server + VERSION>”}
```

<welcome message> is the message the server sends to the client. The server can decide on its contents.

After a while when the client logs the user in:

```
C -> S:  LOGIN_REQ {“username”: “<username>”}
S -> C:  LOGIN_RESP {“status”: “OK”}
```

<username> is the user's username that needs to be logged in.

To other clients (Only applicable when working on Level 2):

```
S -> others: JOINED {"username":"<username>"}
```

Error messages:

```
S -> C: LOGIN_RESP {"status": "ERROR", "code":<error code>}
```

Possible <error code> on LOGIN_RESP:

Error Code	Description
5000	User has already logged in
5001	Username has an invalid format or length
5002	User cannot login twice

2. Broadcast message

Sends a message from a client to all other clients. The sending client does not receive the message himself but gets a confirmation that the message has been sent.

Happy flow:

```
C -> S: BROADCAST_REQ {"message":"<message>"}
S -> C: BROADCAST_RESP {"status": "OK"}
```

<message> is the message that must be sent.

Other clients receive the message as follows:

```
S -> others: BROADCAST {"username":"<username>","message":"<message>"}
```

<username>: is the username of the user sending the message.

Error messages:

```
S -> C: BROADCAST_RESP {"status": "ERROR", "code": <error code>}
```

Possible <error code>:

Error Code	Description
6001	Recipient not found

3. Heartbeat message

Sends a ping message to the client to check whether the client is still active. The receiving client should respond with a pong message to confirm it is still active. If after 3 seconds no pong message has been received by the server, the connection to the client is closed. Before closing, the client is notified with a DSCN message, with reason code 7000. The DSCN is also used if the message is too long to be parsed (reason code 7001). Servers should be capable of parsing messages at least 1024 bytes long.

The server sends a ping message to a client every 10 seconds. The first ping message is sent to the client 10 seconds after the client is logged in.

When the server receives a PONG message while it is not expecting one, a PONG_ERROR message will be returned.

Happy flow:

```
S -> C: PING
C -> S: PONG
```

Error messages:

```
S -> C: DSCN {"reason": <reason code>}
```

[Server disconnects the client]

Possible <reason code>:

Error code	Description
7000	Pong timeout
7001	Unterminated message

```
S -> C: PONG_ERROR {"code": <error code>}
```

Possible <error code>:

Error code	Description
8000	Pong without ping

4. Termination of the connection

When the connection needs to be terminated, the client sends a bye message. This will be answered (with a BYE_RESP message) after which the server will close the socket connection.

Happy flow:

```
C -> S: BYE
S -> C: BYE_RESP {"status": "OK"}
```

[Server closes the socket connection]

Other, still connected clients, clients receive:

```
S -> others: LEFT {"username":"<username>"}
```

Error messages:

None

5. Invalid message header

If the client sends an invalid message header (not defined above), the server replies with an unknown command message. The client remains connected.

Example flow:

```
C -> S: MSG This is an invalid message
S -> C: UNKNOWN_COMMAND
```

6. Invalid message body

If the client sends a valid message, but the body is not valid JSON, the server replies with a parse error message. The client remains connected.

Example flow:

C -> S: BROADCAST_REQ {"aaaa”}
S -> C: PARSE_ERROR

Level 2

7. Listing Connected Users

Show a list of connected clients.

Happy flow:

```
C -> S: LIST_USERS_REQ
S -> C: LIST_USERS_RESP {“status”: “OK”, “users”: [“user1”, “user2”, …, “user n”]}
```

Error messages:

```
S -> C: LIST_USERS_RESP {"status": "ERROR", "code": <error code>}
```

Possible <error code>:

Error Code	Description
6000	User is not logged in

8. Private Message

A user sends a private message to other users.

Happy flow:

```
C -> S: PRIVATE_MESSAGE_REQ {“receiver”: “<username>” , “message”: “<message>”}
S -> C: PRIVATE_MESSAGE_RESP {"status": "OK”}
```

Other clients receive the private message.

```
S -> C2: PRIVATE_MESSAGE {"sender": "<username>", "message": "<message>"}
```

Error messages:

```
S -> C: PRIVATE_MESSAGE_RESP {"status": "ERROR", "code": <error code>}
```

Possible <error code>:

Error Code	Description
6000	User is not logged in
6001	Recipient not found
6007	Cannot send messages to yourself
6008	Empty message body

9. Guessing Game

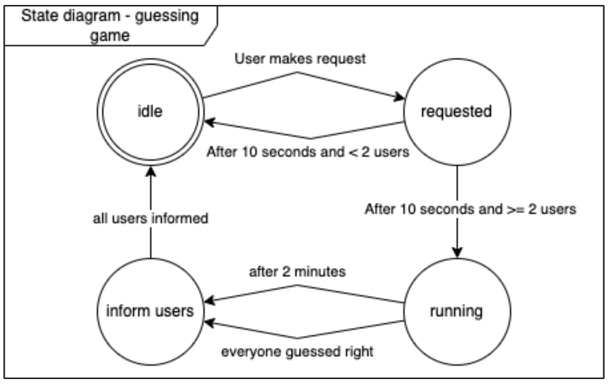

Figure 1: Guessing Game Lifecycle

9.1. User initiates the game.

Happy flow:

```
C -> S: START_GAME_REQ
```

[Server receives the request and begins the game setup]

```
S -> C: START_GAME_RESP {"status": "OK" }
```

[Server waits 10 seconds for users to join]

Other connected users receive a notification about the ongoing game.

```
S -> C2: GAME_NOTIFICATION {“status”: “OK”}
```

Users join the game.

```
C2 -> S: JOIN_GAME_REQ
```

The server acknowledges the user’s participation.

```
S -> C2: JOIN_GAME_RESP {“status”: “OK”}
```

After 10 seconds and >= 2 users, the game starts.

```
S -> C: START_GAME_RESP {“status”: “OK”}
S -> C2: START_GAME_RESP {“status”: “Ok”}
```

Error messages:
After 10 seconds and <2 users, the game is cancelled.

```
S -> C: START_GAME_RESP {"status": "ERROR", "code": <error code>}
```

Possible <error code>:

Error Code	**Description**
6000	User is not logged in.
6001	Recipient not found.
6002	Insufficient players to start the game.
6003	Game has already started; cannot join.
6004	User already joined.
6005	No running game.

9.2. Making a Guess

[Server selects a number between 1 and 50]

Happy flow:

```
C -> S: GUESS_NUMBER_REQ {“number”: “<guessed number>”}
S -> C: GUESS_NUMBER_RESP {"status": "OK", "result": "<comparison result>"}
```

User’s guess is correct.

S -> C2: GUESS_NUMBER_RESP {"status": "OK", "result": "Correct"}

Error messages:

S -> C: GUESS_NUMBER_RESP {“status”: “ERROR”, “code”: <error code>}

Possible <error code>

Error Code	Description
6000	User is not logged in.
6003	Game has already started; cannot join.
7002	User did not receive the notification.
7003	Game is ongoing; cannot make a guess.
7004	Invalid guess format.
7005	Your guess is too low.
7006	Your guess is too high.
7007	Number out of allowed range.

9.3. Informing users

[Server informs each user of their progress while guessing]

Happy flow:

S -> C: GUESS_NUMBER_RESP {“message”: “<message>”}

Error messages:

S -> C2: GUESS_NUMBER_RESP {“status”: “ERROR”, “code”: <error code>}

Possible <error code>

Error Code	Description
8001	User is not participating.

9.4. End game.

[Server calculates the guessing time for all users and publishes the results (asc)]

Show the results.

S -> C: GAME_RESULTS {“status”: “OK”, “results”: “<sorted results list>”}

Error messages:

S -> C: GAME_RESULTS {“status”: “ERROR”, “code”: <error code>}

Possible <error code>

Error Code	Description
9002	No game results are available.

9.5. Guessing Game JSON Structure Example

Server logs:

```
[
  {
    “badr” --> START_GAME_REQ
    “brice” --> JOIN_GAME_REQ
    “brice has joined the game.”
  },
  {
    “badr” --> GUESS_NUMBER_REQ {“number”: 45},
    “badr” <-- GUESS_NUMBER_RESP {"result”: -1}
  },
  {
    “brice” --> GUESS_NUMBER_REQ {“number”: 50},
    “brice” <-- GUESS_NUMBER_RESP {“result”: 1}
  },
  {
    “badr” --> GUESS_NUMBER_REQ {“number”: 55}

  },
  {
    “brice” --> GUESS_NUMBER_REQ {“number”: 47},
    “brice” <-- GUESS_NUMBER_RESP {"result": 0}
  },
  {
    “badr” --> GUESS_NUMBER_REQ {“number”: 47},
    “badr” <-- GUESS_NUMBER_RESP {“result”: 0}
  },
  {
    “brice” <-- GAME_RESULTS {“brice”:”13744 ms”,”badr”:”17859 ms”},
    “badr” <-- GAME_RESULTS {“brice”:”13744 ms”,”badr”:”17859 ms”}
  }
 ]
```

Client logs:

Badr:

```
{
 "Choose 5"
"message": "Game initiation successful. Waiting for
players to join..."
"message": "You have joined the game."
},

{
 "You have joined the game."
"--------------------
 The game has started with the following participants:
  - badr
  - brice
  --------------------
  You can now make a guess between 1 & 50. Good
Luck! "
},

{
"Choose 7"
"Enter your guess:  45"
"Guessed Number: 45"
"Your guess is too low."
},

{
"Choose 7"
"Enter your guess:  55"
"Your guess is out of range! Make a guess between 1 &
50."
},

{
"Choose 7"
"Enter your guess:  47"
"Guessed Number: 47"
Congratulations! Your guess is correct!
----------------------------------------
Guessing Game Results:
1. brice (winner, 13744 ms)
2. badr (17859 ms)
----------------------------------------
}
```

Brice:

```
{
  "message": "A guessing game has been initiated. Join
now!"
 "Choose 6",
 "message": "You have joined the game."
},

{
 "You have joined the game."
"--------------------
 The game has started with the following participants:
  - badr
  - brice
  --------------------
  You can now make a guess between 1 & 50. Good
Luck! "
},

{
"Choose 7"
"Enter your guess:  50"
"Guessed Number: 50"
"Your guess is too high."
},

{
"Choose 7"
"Enter your guess:  47"
"Guessed Number: 47"
Congratulations! Your guess is correct!
----------------------------------------
Guessing Game Results:
1. brice (winner, 13744 ms)
2. badr (17859 ms)
----------------------------------------
}
},
```

Level 3

10. File Transfer

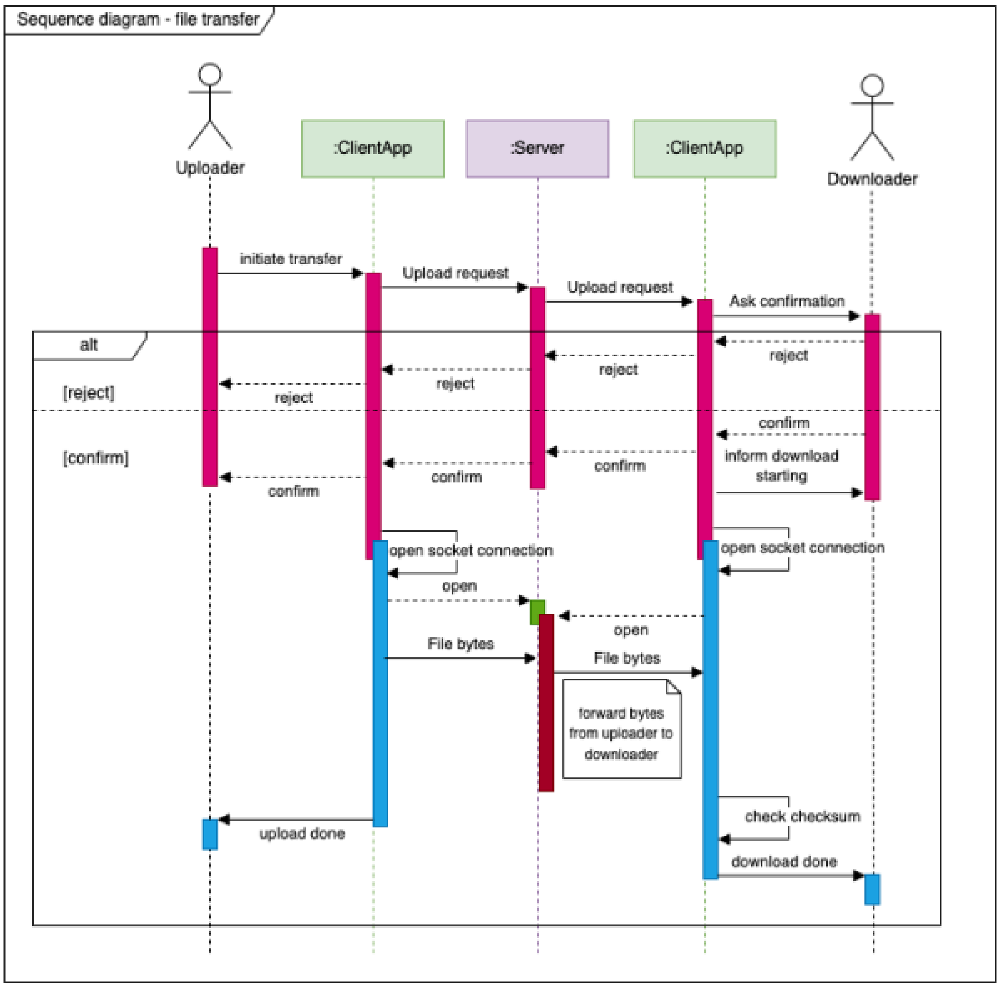

Figure 2 General overview file-transfer

10.1. Request File Transfer (Upload process)

Happy flow

Uploader -> S: FILE_TRANSFER_REQUEST {“receiver”: “<receiver_username>”, “filename”: “<file_name>"}

S -> downloader: FILE_TRANSFER_REQUEST {“sender”: “<sender_username>”, “filename”: “<file_name>"}
Downloader -> S : FILE_TRANSFER_RESPONSE {“sender”: “<sender_username>”, “status”: “<status>", “transferId” : “<TransferUUID>”}

S -> Uploader: FILE_TRANSFER_RESPONSE {“receiver” : “<receiver_username>”, “status”: “<status>”, “transferId” : “<TransferUUID>”}

Error messages:

S -> Uploader: FILE_TRANSFER_RESPONSE {“status”: “ERROR”, “code”: <error code>}

Possible <error code>

Error Code	Description
6000	User is not logged in.
6001	Recipient not found.
10000	File size exceeds available memory.
10001	Invalid checksum format.
10002	Receiver has rejected the file.

The rest of the downloading and uploading are outside the protocol as the teacher suggested.

Level 4

11. Encryption message

First, client C1 initiates by requesting to send an encrypted message to another client C2.

C1 -> S: SECURE_MESSAGE_REQ {“receiver”: “<username>”}

Other clients receive the handshake request and send their public KEY

```
S -> C2: SECURE_MESSAGE_REQ {"sender": "<username>"}
C2 -> S: SECURE_MESSAGE_RES {“sender”:”<username>”, “publicKey”: “<publicKey>”}
```

The sender C1 receives the public key and sends a session key encrypted using the public key, and a message encrypted using the session key.

```
S -> C1: SECURE_MESSAGE_RES {"receiver": "<username>", “publicKey”: “<publicKey>”}
C1 -> S: SECURE_MESSAGE {“sender”:<username>, “sessionKey”: “<sessionKey>”, “message” :
“<encryptedmessage>”}
```

Other clients receive the private encrypted message and the encrypted session key.

```
S -> C2: SECURE_MESSAGE {"sender": "<username>", “sessionKey”: ”<sessionKey>”, “message”:
“<encryptedMessage>”}
```

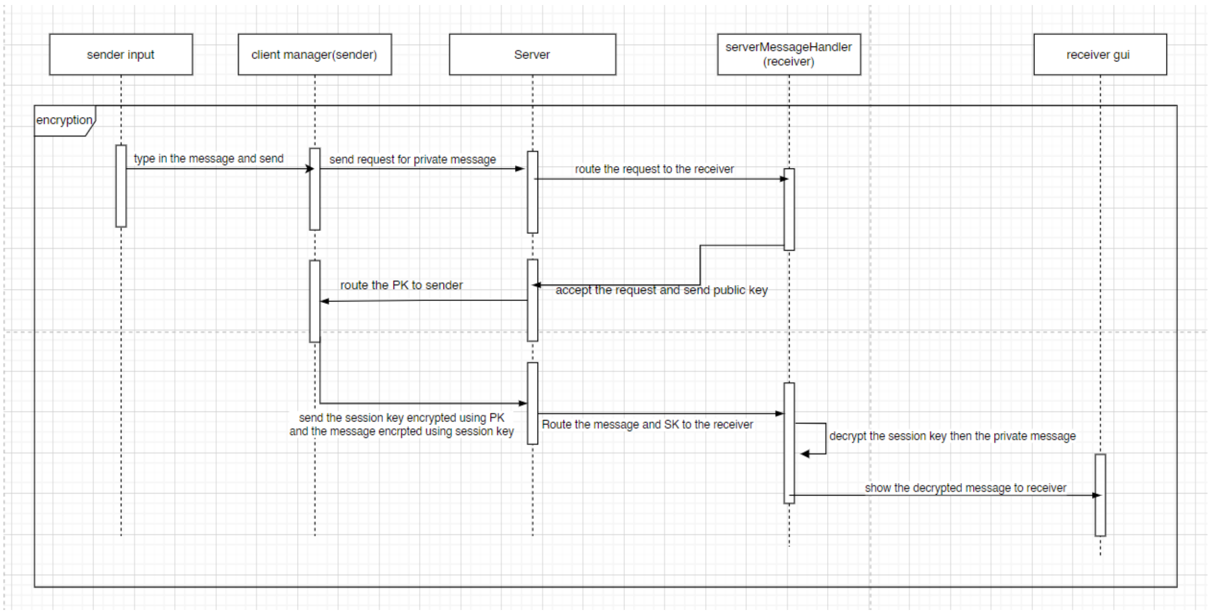
sender input
client manager(sender)
Server
serverMessageHandler (receiver)
receiver gui
encryption
type in the message and send
send request for private message
route the request to the receiver
route the PK to sender
accept the request and send public key
send the session key encrypted using PK and the message encrpted using session key
Route the message and SK to the receiver
decrypt the session key then the private message
show the decrypted message to receiver

Figure 2: Sequence diagram of encryption.