



CG4002 Embedded System Design Project

April 2022 semester

“SHOOT_IT” Design Report

Group -	Name	Student #	Specific Contribution
Member	Lee Juntong	-	Comms External & Sw Visualizer

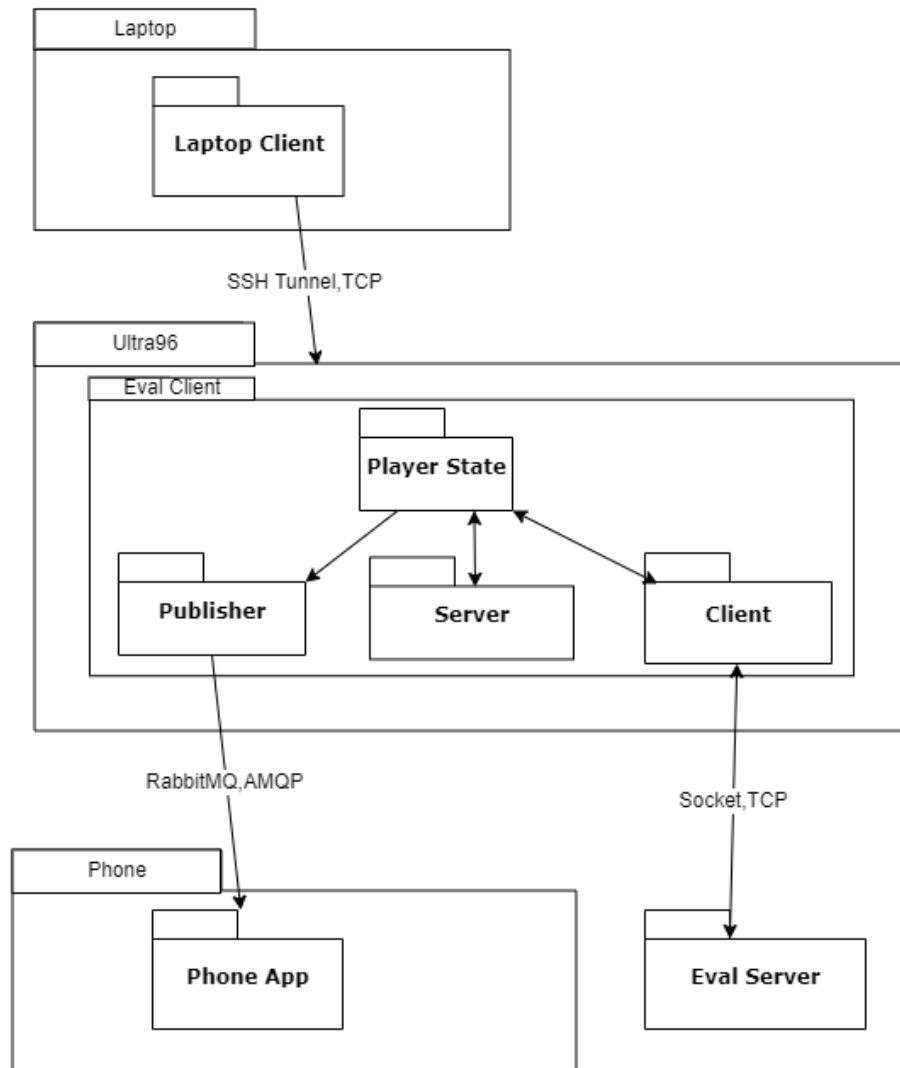
Section 1 System Functionalities

Feature List

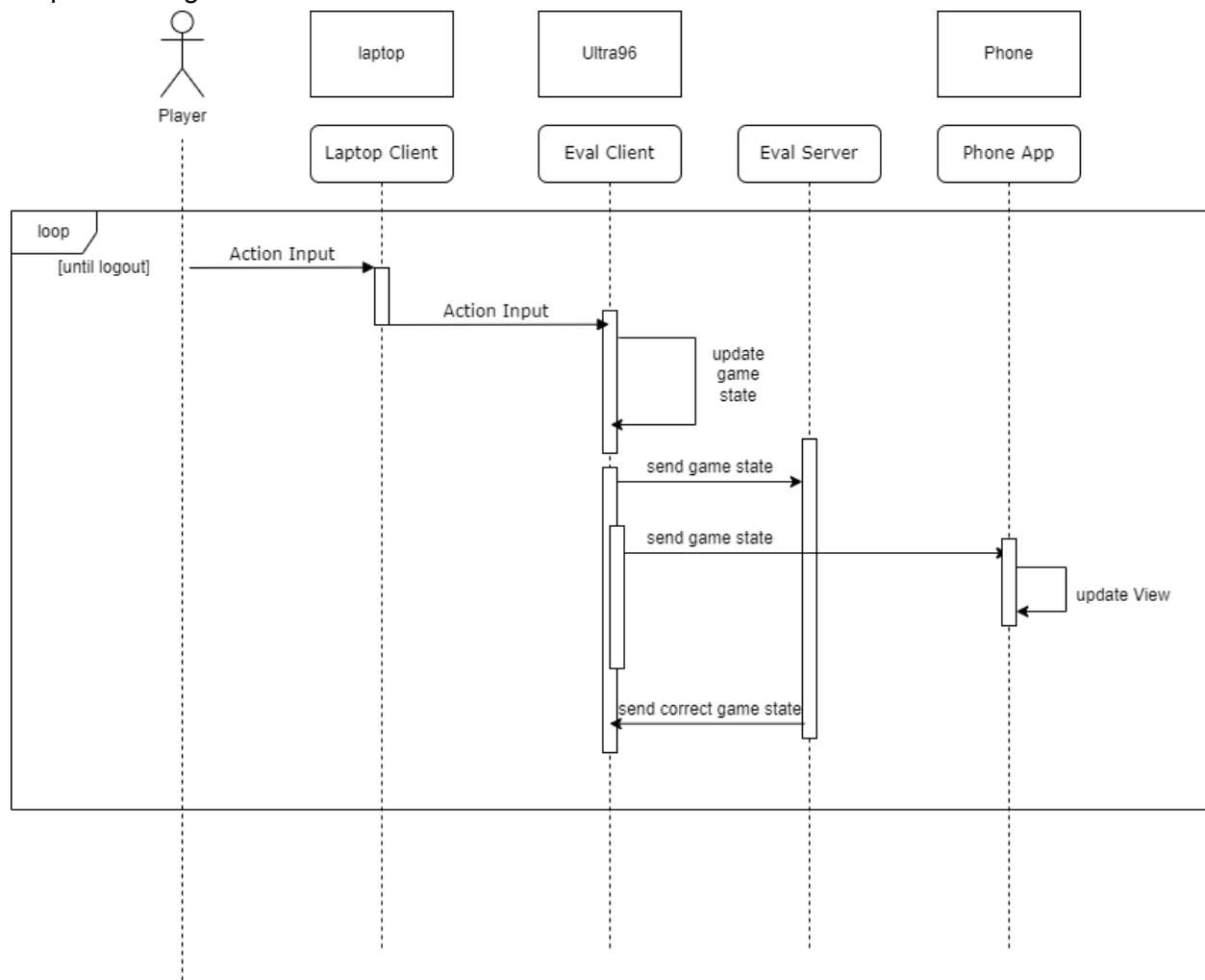
Feature
grenade: Throw a grenade to the opponent to deal 30 dmg, if the player has at least 1 grenade.
shoot: Shoot the opponent to deal 10 dmg, if the player has at least 1 bullet.
shield: Create a shield that lasts 10s to block at most 30 dmg, if the player has at least 1 shield and has not created a shield in the last 10s.
reload: If the player has 0 bullets, reload to 6 bullets.
pos [0~4]: Move the player to the corresponding position, where 0 represents disconnected, 4 represents a shelter where player won't be hit by opponent who is not at position 4
logout: Stop receiving new actions

Section 2 Overall System Architecture

overall architecture



Sequence Diagram



There are 3 types of threads running in Eval Client.

Publisher: publish the current game state to the corresponding queue on rabbitmq server, for phone app to retrieve.

Server: take in the action input from the laptop client, update the game state accordingly.

Client: send the current game state to the eval server, then update to the correct/expected one sent back by the eval server.

For single player mode, there will be 1 thread for each type.

For 2 player mode, there will be 2 publisher threads (correspondingly, 2 queues on rabbitmq server), 2 server threads (each one handles one player), 1 client thread.

Section 3 External Communications

Communicate between Ultra96 and the evaluation server.[1][2][3]

```
self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_address = (ip_addr, port_num)
self.socket.connect(server_address)
```

In eval_client.py, client thread is the thread communicating with Ultra96. First, it creates a TCP socket with the IP address and port given, then connects to the eval server.

```
while not _shutdown.is_set():
    event_send_to_server.wait()
    state.send_encrypted(self.socket, self.key)
    state.recv_and_update(self.socket)
    event_send_to_server.clear()
```

In this thread, whether to communicate with the eval server is controlled by the event, event_send_to_server.

Secure communications are ensured when sending to the eval server.

```
# encrypt and send the game state json to remote host
def send_encrypted(self, remote_socket, secret_key_string):
    success = True
    plaintext = self._get_data_plain_text()

    ice_print_debug(f"Sending message to server: {plaintext} (Unencrypted)")
    plaintext_bytes = pad(plaintext.encode("utf-8"), 16)

    secret_key_bytes = secret_key_string.encode("utf-8")
    cipher = AES.new(secret_key_bytes, AES.MODE_CBC)
    iv_bytes = cipher.iv

    ciphertext_bytes = cipher.encrypt(plaintext_bytes)
    message = base64.b64encode(iv_bytes + ciphertext_bytes)

    # send len followed by '_' followed by cypher
    m = str(len(message)) + '_'
    try:
        remote_socket.sendall(m.encode("utf-8"))
        remote_socket.sendall(message)
    except OSError:
        print("Connection terminated")
        success = False
    return success
```

AES (Advanced Encryption Standard) is used for encryption.

Start by getting the data in plain text, which is exactly the format of message that the eval server wants after decoding and decryption.

```
def _get_data_plain_text (self):  
    data = {'p1': self.player_1.get_dict(), 'p2': self.player_2.get_dict()}  
    return json.dumps(data)+'#'
```

It is a json string containing the dictionaries which contains both players' states, with a '#' at the end. There is no specific reason for '#' other than accommodating the format specified in the eval server.

Then encode the string with utf-8, set the block boundary to be 16, pad the byte string accordingly. 16 is chosen due to AES works on blocks of size 16 byte.

Afterwards, encode the secret key to the byte string, choose AES mode to be Ciphertext Block Chaining, and create a cipher(an encryption algorithm) based on them. Get the initialization vector, which will be used to provide the initial state and for the server to decrypt the message.

Now, encrypt the data to byte string, concatenate it to iv, and encode them with Base64.

Send length of message followed by '_' followed by previously encoded message, which is specified in eval server. This finishes sending a message to the eval server.

Receive and update, on the other hand, receives messages from the eval server that are not encrypted. These messages contain the expected game state. This function will decode the message received, and process it to become a json object that contains the game state, based on the format specified by the eval server. It then updates the game state.

```
from Crypto.Cipher import AES  
from Crypto.Util.Padding import pad  
import base64
```

These are the libraries used, which correspond to creating new cipher, padding, and encoding with Base64 respectively.

Communicate between laptops and Ultra96. [3][4]

One can use ssh tunneling via sunfire to allow the communication between the laptop and the ultra96.

The following commands are inputted in localhost, where -L stands for Local Forwarding, which means forwarding a port from the client machine to the server machine.

```
ssh -L 8899:localhost:8899 <user_id>@sunfire.comp.nus.edu.sg
```

```
ssh -L 8898:localhost:8898 <user_id>@sunfire.comp.nus.edu.sg
```

Then in sunfire, bind the ports to ultra96 again.

```
ssh -L 8899:localhost:8899 xilinx@137.132.86.233
```

```
ssh -L 8898:localhost:8898 xilinx@137.132.86.233
```

Note that these shall be done in 2 terminals, and the terminals can't be closed so the binding is not lost.

One may use `ssh -f -N -L`, so it can be done in one terminal.

That is, in localhost,

```
ssh -f -N -L 8899:localhost:8899 <user_id>@sunfire.comp.nus.edu.sg
```

```
ssh -f -N -L 8898:localhost:8898 <user_id>@sunfire.comp.nus.edu.sg
```

in sunfire,

```
ssh -f -N -L 8899:localhost:8899 xilinx@137.132.86.233
```

```
ssh -f -N -L 8898:localhost:8898 xilinx@137.132.86.233
```

Closing this terminal leads to the binding being lost. They are equivalent and both methods work fine.

8899 is the port for player 1 and 8898 is the port for player 2. They are hard coded so one does not need to specify the ports to connect to laptop clients when starting the eval client. The code can be easily changed to accept arguments for the port numbers and allow the ports to be specified upon starting the eval client.

After binding the ports, one can send messages from laptop clients to the eval client which runs in the ultra96.

```
while True:
    action=input("action:")
    my_client.send_message(action)
    if action=="logout":
        break
```

```
# To send the message to the sever
def send_message(self, message):
    encrypted_message = self.encrypt_message(message)
    self.socket.sendall(encrypted_message)
```

It keeps taking user input, until a logout command is issued.

It sends an encrypted message, which is encrypted in a similar way as the eval client communicates with the eval server. The difference is, this time the message is just a short string, containing the action of the player.

It's then received by the server thread in the eval client.

```

self.socket.listen(1)

self.client_address, self.secret_key = self.setup_connection()    # Wait for secret key

while not self.shutdown.is_set():    # Stop waiting for data if we received a shutdown signal
    data = self.connection.recv(1024)    # Blocking wait for data
    update_pos_flag=False
    if data:
        try:
            msg = data.decode("utf8")    # Decode raw bytes to UTF-8
            decrypted_message = self.decrypt_message(msg)    # Decrypt message using secret key
            # If no valid action was sent
            if len(decrypted_message) == 0:
                pass
            else:
                self.has_no_response = False
                action=decrypted_message
                print(f"received: player{self.player_index}:{action}")


```

After decoding and decryption (almost the same as the corresponding functions in the eval server), we retrieve the action by this player.

```

class Server(threading.Thread):
    def __init__(self, ip_addr, port_num, player_index):
        super(Server, self).__init__()
        self.player_index=player_index

```



Note that when creating a server thread, the player index is required, so that the thread knows which player is doing the action.

Afterwards, in this thread, it updates the game state with respect to the rules specified.

Rules (No Change)

- Health Point
 - Player: 100 HP.
 - Shield: 30 HP/10 Sec
- Damage
 - Bullet: 10 HP
 - Grenade: 30 HP

Rules (No Change)

- Ammo and constraints
 - Unlimited magazines
 - 6 bullets per magazine
 - Reload can be performed only if the magazine is empty
 - 2 grenades per life
 - 3 shields per lifetime
 - Cannot activate shield within 10 sec of previous activation, even if 0-HP

```

class PlayerStateStudent(PlayerStateBase):
>     def action_is_valid(self, action_self): ...
>     def update(self, pos_self, pos_opponent, action_self, ...

```

It is done through these 2 functions.

In summary, the communication is very similar to the communication between the eval client and the eval server, the only thing one needs to do is to bind the ports so the laptop can send messages to ultra96.

Handle concurrency on the laptops or Ultra96.[3][5]

For the laptop, in 2 players mode, open 2 terminals to run 2 instances of the python script laptop_client.py deals with concurrency.

For the Ultra96, create 3 or 5 threads, depending on whether it is single player mode.

The threads are controlled by events. When event.wait() is called, the thread stops until the event is set, and while waiting, the resources can be used by other threads, and hence handles concurrency.

```

event_send_to_server=threading.Event()
event_send_to_phone1=threading.Event()
event_send_to_phone2=threading.Event()

```

In particular, event_send_to_server is used to control client thread, which talks to eval server.

```

event_send_to_server.wait()
state.send_encrypted(self.socket,self.key)
state.recv_and_update(self.socket)
event_send_to_server.clear()

```

send_to_phone1/2 are used to control publisher threads, which send messages to queues on RabbitMQ server.

```

def run(self):
    global state

    while True:
        if self.player==1:
            event_send_to_phone1.wait()
        else:
            event_send_to_phone2.wait()
        self.channel.basic_publish(exchange='', routing_key=self.queueName, body=state._get_data_plain_text_phone())
        print("sending message in rabbitmq:")
        print(state._get_data_plain_text_phone())
        if self.player==1:
            event_send_to_phone1.clear()
        else: event_send_to_phone2.clear()
        if _shutdown.is_set():
            self.connection.close()
            sys.exit()

```

For server thread,

```

data = self.connection.recv(1024) # Blocking wait for data

```

blocking wait for data allows the use of resources by other threads when there is no data coming.

Communicate between Ultra96 and Visualizer.[5][6]

Publisher threads are responsible for this communication.

```

class Publisher(threading.Thread):
    def __init__(self,queueName,player):
        super(Publisher, self).__init__()
        self.player=player#indicates player 1 or 2
        self.queueName=queueName
        credentials = pika.PlainCredentials('admin1', 'admin123')#username,password
        self.connection = pika.BlockingConnection(pika.ConnectionParameters(pc_ip_address,5672,'/',credentials))
        self.channel = self.connection.channel()
        self.channel.queue_declare(queue=self.queueName)

    def run(self):
        global state

        while True:
            if self.player==1:
                event_send_to_phone1.wait()
            else:
                event_send_to_phone2.wait()
            self.channel.basic_publish(exchange='', routing_key=self.queueName, body=state._get_data_plain_text_phone())
            print("sending message in rabbitmq:")
            print(state._get_data_plain_text_phone())
            if self.player==1:
                event_send_to_phone1.clear()
            else: event_send_to_phone2.clear()
            if _shutdown.is_set():
                self.connection.close()
                sys.exit()

```

When the thread is created, it connects to the RabbitMQ server. In this case, the server is running on my desktop. Hence, pc_ip_address is my desktop's ip address. The

credentials is a tuple of username and password for an administrative account of this RabbitMQ server.

When the event_send_to_phone is set, it will publish a message to the queue in the RabbitMQ server.

```
self.channel.basic_publish(exchange='', routing_key=self.queueName, body=state._get_data_plain_text_phone())
```

I have no exchange, so exchange="". For routing_key, the queue name is expected to be the argument, so it knows which queue should be taking the message. The body is the message being sent, which is the json string describing the game state. This string has not "#" at the end, so one less step for processing in the phone app.

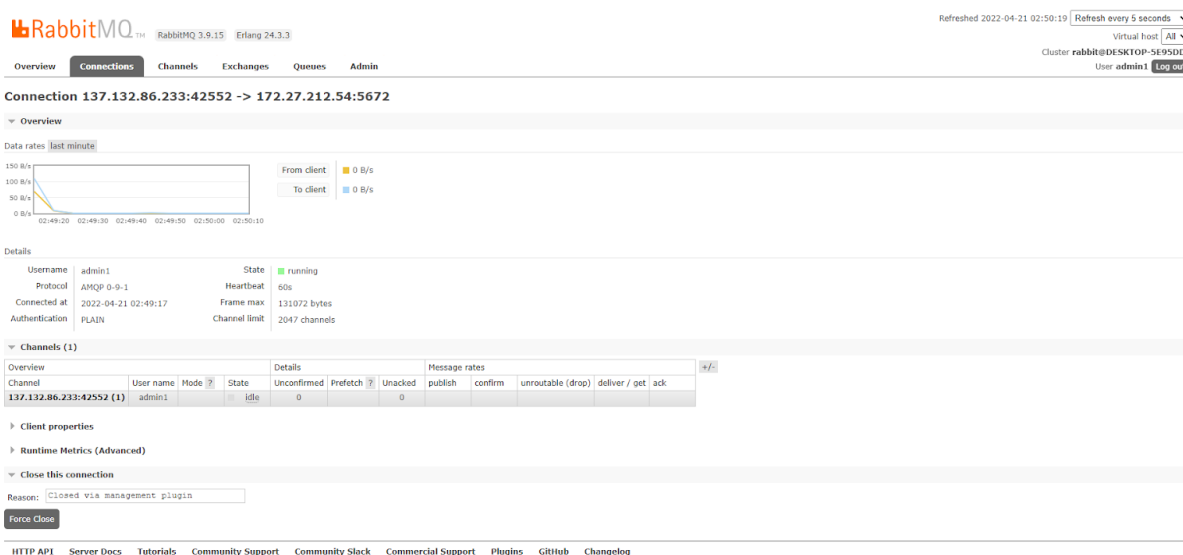
When the player is player 1, it will publish the message to queue "phone1", which has a consumer, the visualizer/phone app. The phone app for player 1 will consume the message from the queue and then process it. The phone app is connected to the RabbitMQ server as a consumer.

Player 2 is similarly handled.

The visualizer is not sending anything back to ultra96.

Detour to RabbitMQ server[7][8][9]

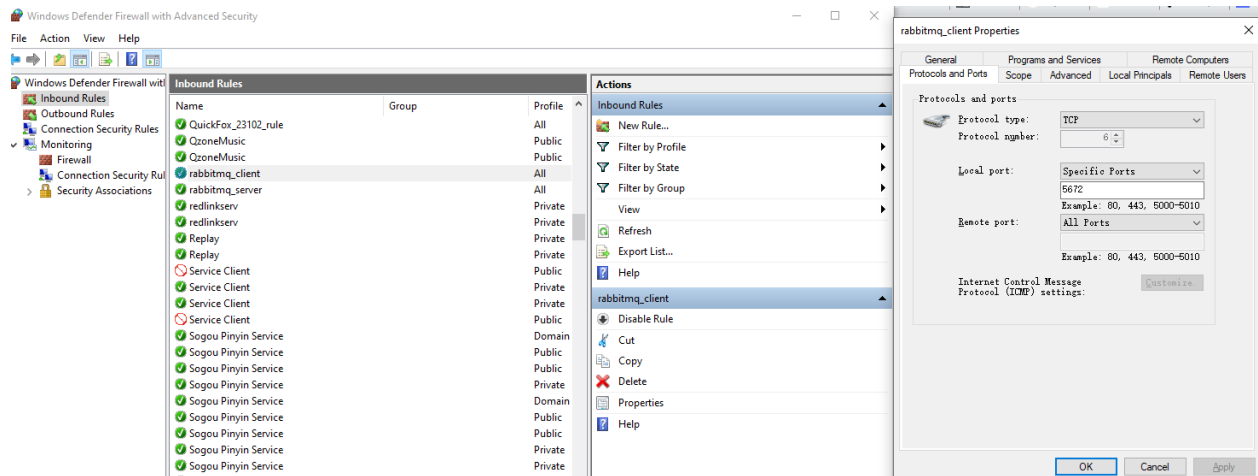
In ultra96, it can connect to port 5672 on my desktop, where the RabbitMQ server is running.



The connectivity can be checked by telnet <ip_address> <port_number>

```
xilinx@pynq:~/CG4002$ telnet 172.27.212.54 5672
Trying 172.27.212.54...
Connected to 172.27.212.54.
Escape character is '^J'.
Connection closed by foreign host.
xilinx@pynq:~/CG4002$
```

This is because the port 5672 is open on my desktop.



Here is an instruction for opening a port on Windows 10.

Follow the steps below and check if you are able to open a portal in Firewall on the computer.

1. Press **Windows logo + X** keys on the keyboard and select **Control panel** from the context menu.
2. Select **System and Security** from the options and click on **Windows Firewall** from right side panel of the window.
3. Click on **Advanced settings** and select **Inbound Rules** from left side panel of the window.
4. Click on **New Rule** under **Actions** tab from right side panel and select **Port** radio button from the window.
5. Follow the onscreen instructions and check if the changes are effective.

However, simply pinging the ip address (by using the command `ping ip_address`) of the PC will not work, as not all ports are open.

After installing RabbitMQ server, one can run the following in the `sbin` folder to start the server. The instruction on RabbitMQ website is outdated/not suitable for Windows 10.

```
set RABBITMQ_BASE=<data_file_directory>
set RABBITMQ_CONFIG_FILES=<configuration_files_directory>
rabbitmq-server.bat
```

These commands set the data directory, the configuration files directory, then directly start the server. Do not start the service first, so some bugs can be avoided.

```
Administrator: Command Prompt - rabbitmq-server.bat
D:\rabbitmq\rabbitmq_server-3.9.15\sbin>set RABBITMQ_BASE=D:\rabbitmq\rabbitmq_server-3.9.15\data
D:\rabbitmq\rabbitmq_server-3.9.15\sbin>rabbitmq-server.bat
2022-04-21 02:38:17.080000+08:00 [info] <0.228.0> Feature flags: list of feature flags found:
2022-04-21 02:38:17.092000+08:00 [info] <0.228.0> Feature flags: [x] implicit_default_bindings
2022-04-21 02:38:17.092000+08:00 [info] <0.228.0> Feature flags: [x] maintenance_mode_status
2022-04-21 02:38:17.092000+08:00 [info] <0.228.0> Feature flags: [x] quorum_queue
2022-04-21 02:38:17.092000+08:00 [info] <0.228.0> Feature flags: [x] stream_queue
2022-04-21 02:38:17.092000+08:00 [info] <0.228.0> Feature flags: [x] user_limits
2022-04-21 02:38:17.092000+08:00 [info] <0.228.0> Feature flags: [x] virtual_host_metadata
2022-04-21 02:38:17.092000+08:00 [info] <0.228.0> Feature flags: feature flag states written to disk: yes
2022-04-21 02:38:17.696000+08:00 [noti] <0.44.0> Application syslog exited with reason: stopped
2022-04-21 02:38:17.696000+08:00 [noti] <0.228.0> Logging: switching to configured handler(s); following messages may no
t be visible in this log output

## ##      RabbitMQ 3.9.15
## ##
##### Copyright (c) 2007-2022 VMware, Inc. or its affiliates.
##### ##
##### Licensed under the MPL 2.0. Website: https://rabbitmq.com

Erlang:      24.3.3 [jit]
TLS Library: OpenSSL - OpenSSL 1.1.1d 10 Sep 2019

Doc guides:  https://rabbitmq.com/documentation.html
Support:     https://rabbitmq.com/contact.html
Tutorials:   https://rabbitmq.com/getstarted.html
Monitoring:  https://rabbitmq.com/monitoring.html

Logs: <stdout>
      d:/rabbitmq/rabbitmq_server-3.9.15/data/log/rabbit@DESKTOP-5E95DDI.log
      d:/rabbitmq/rabbitmq_server-3.9.15/data/log/rabbit@DESKTOP-5E95DDI_upgrade.log

Config file(s): (none)
```

It will look like this if the server starts successfully.

Section 4 Software Visualizer

The visualizer design[11]

CG4002 SHOOT GAME

Player 1

hp: 100
action: none
bullets: 6
grenades: 2
shield_time: 0
shield_health: 0
num_deaths: 0
num_shield: 3

Player 2

hp: 100
action: none
bullets: 6
grenades: 2
shield_time: 0
shield_health: 0
num_deaths: 0
num_shield: 3

Both player 1's and player 2's detailed information will be displayed. I suppose this is known as "know yourself and the enemy".

Visualizer software architecture[6][10][12]

rabbitmq.client and json.simple are used.

```
public void receiving(ActivityMainBinding binding) {
    consumerThread = new Thread(new Runnable() {
        @Override
        public void run() {
            setupConnection();

            DeliverCallback deliverCallback = (consumerTag, delivery) -> {
                String message = new String(delivery.getBody(), StandardCharsets.UTF_8);
                try {
                    parse_update(message, binding);
                } catch (JSONException e) {
                    e.printStackTrace();
                } finally {
                    channel.basicAck(delivery.getEnvelope().getDeliveryTag(), multiple: false);
                }
            };
            try {
                channel.basicConsume(QueueName, autoAck: false, deliverCallback, consumerTag -> {
                });
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    });
    consumerThread.start();
}
```

When creating the main activity, the function receiving() is called, which starts a separate thread to deal with receiving information from the RabbitMQ server.

In deliverCallback, it takes in items in the queue one by one and updates the view based on the information received.

It also sends acknowledgement to the server only if the information is successfully received.


```

private void parse_update(String message, ActivityMainBinding binding) throws JSONException {
    JSONObject player_status = (JSONObject) JSONValue.parse(message);
    HashMap p1 = (HashMap) player_status.get("p1");
    HashMap p2 = (HashMap) player_status.get("p2");
    String p1_hp = p1.get("hp").toString();
    String p1_action = p1.get("action").toString();
    String p1_bullets = p1.get("bullets").toString();
    String p1_grenades = p1.get("grenades").toString();
    String p1_shield_time = p1.get("shield_time").toString();
    String p1_shield_health = p1.get("shield_health").toString();
    String p1_num_deaths = p1.get("num_deaths").toString();
    String p1_num_shield = p1.get("num_shield").toString();
    String p2_hp = p2.get("hp").toString();
    String p2_action = p2.get("action").toString();
    String p2_bullets = p2.get("bullets").toString();
    String p2_grenades = p2.get("grenades").toString();
    String p2_shield_time = p2.get("shield_time").toString();
    String p2_shield_health = p2.get("shield_health").toString();
    String p2_num_deaths = p2.get("num_deaths").toString();
    String p2_num_shield = p2.get("num_shield").toString();
    String p1_display = "hp: " + p1_hp + "\naction: " + p1_action + "\nbullets: " + p1_bullets + "\ngrenades: " + p1_grenades + "\nshield_time: " + p1_shield_time + "\nshield_health: " + p1_shield_health + "\nnum_deaths: " + p1_num_deaths + "\nnum_shield: " + p1_num_shield;
    String p2_display = "hp: " + p2_hp + "\naction: " + p2_action + "\nbullets: " + p2_bullets + "\ngrenades: " + p2_grenades + "\nshield_time: " + p2_shield_time + "\nshield_health: " + p2_shield_health + "\nnum_deaths: " + p2_num_deaths + "\nnum_shield: " + p2_num_shield;
    Runnable runnable = new Runnable() {
        @Override
        public void run() {
            binding.player1Status.setText(p1_display);
            binding.player2Status.setText(p2_display);
        }
    };
    runOnUiThread(runnable);
}

```

The message being sent from the Publisher thread in the eval client is a json string. It can be parsed to a JSONObject using the parse function in the library json.simple. A JSONObject is similar to a dictionary in python. one can use get(key) to get the object with this key.

Since the original dictionary is 2 dimensional, get() will provide an object with the shape of a dictionary. This is HashMap in JAVA. Cast it to HashMap, store it, then the attributes can be retrieved one by one.

After having all the data, one can set up the display.

binding is a ViewBinding object, allowing the user to control views without the use of findViewById(). player1Status is a TextView, and one can use setText to edit the string being displayed in this TextView.

The update is in a Runnable object, runnable. It's then passed to runOnUiThread() to update the view.

In android, one can only update the UI on the main thread. So one can either choose to use a handler, or simply call runOnUiThread() and pass in the update to be run.

Section 5 Societal and Ethical Impact

When talking about wearables, one of the most attractive products is the google glass. In the future, we may have a far more powerful substitute, say google glass pro plus max ultra. This new product may be able to detect eye movement and based on that, find where the person wearing it is looking at. If the person is looking at some object, a brief description regarding it will be displayed, and one can either choose to close it or dig in deeper. Based on what is the object, further actions can defer. For instance, if it's some common goods, a link for buying it may be available. When the person's contacts are getting focused, the person will have the information they recorded regarding the contacts, so as to avoid forgetting some information that can be useful, for example, their friends' birthday. This brings convenience to people.

On the other hand, AR wearables may lead to ethical concerns, just as all other technologies that make people's life easier. Still using google glass pro plus max ultra as an example, it's now much easier to find out what kind of goods that one prefers, as the wearable will get much more data regarding what kind of goods one focuses on more. If google uses this data to create customized advertisements, it threatens people's privacy.

For balancing, laws and regulations are required, in order to prevent the abuse of technologies which lead to ethical problems. However, monitoring how the data being used will be more important, otherwise laws can't be enforced. To do so, the government may need to put manpower to the firms. Meanwhile, who is responsible for supervising the government? When others have more data regarding a person, their privacy can be threatened, unless they are able to find ways to prevent more data regarding them being processed, including sending fake data to the server. However, by doing so, such technologies bring inconvenience to people, which is against its will.

References

[1] Using AES [Online].

Available: <https://pycryptodome.readthedocs.io/en/latest/src/cipher/aes.html>

[2] Using json in python [Online].

Available: <https://docs.python.org/3/library/json.html>

[3] Using sockets in python [Online].

Available: <https://docs.python.org/3/library/socket.html>

[4] SSH port forwarding - Example, command, server config [Online].

Available: <https://www.ssh.com/academy/ssh/tunneling/example>

[5] Using threading in python [Online].

Available: <https://docs.python.org/3/library/threading.html>

[6] RabbitMQ tutorial [Online]. Available: [RabbitMQ tutorial - "Hello world!" — RabbitMQ](#)

[7] RabbitMQ server installation [Online].

Available: <https://www.rabbitmq.com/download.html>

[8] Opening ports in Windows10 [Online].

Available: <https://answers.microsoft.com/en-us/windows/forum/all/how-to-open-port-in-windows-10-firewall/f38f67c8-23e8-459d-9552-c1b94cca579a>

[9] Solving RabbitMQ server not running [Online]. Available: [rabbitmq-server.bat 启动闪退-asfx社区](#)

[10] Using json in JAVA [Online].

Available: <https://www.geeksforgeeks.org/parse-json-java/>

[11] Using ViewBinding for android [Online]. Available: [View Binding | Android Developers](#)

[12] Updating UI in android [Online]. Available: [ui thread - How do we use runOnUiThread in Android? - Stack Overflow](#)