Requirement 1 - Encryption:

The public key system is fairly straightforward; on startup, clients generate their own public and private keys,

```
def main(argv):
...

      private_key = rsa.generate_private_key(public_exponent=65537,
key_size=2048, backend=default_backend())
      public_key = private_key.public_key()

      connect(public_key, name, port) #register with the server
```

and then forwards its public key, as well as its assigned name and port to the server.

```
def connect(key, name, port):
   key_bytes = key.public_bytes(encoding=serialization.Encoding.PEM,
format=serialization.PublicFormat.SubjectPublicKeyInfo)
   body = {"name": name, "port": port, "public_key": key_bytes} #name,
listening socket port and public key; this is everything the server needs to
manage each client
   while True:
       response = requests.post(("http://127.0.0.1:" + str(SERVER_PORT) +
"/connect"), data=body)

       if response.status_code == 200:
           print(response.text, "\n") #successful connection
           return
       else:
           print("Failed to connect. error:\n", str(response.text), "\n\n")
           time.sleep(5) #if we fail to connect, we wait 5 seconds before
trying to connect again
```

On the server side, when the connect method is called by a client, the server extracts the name, port and public key; all the info required to connect, from the request data, and stores it in 2 dicts; first, “members”, where the port number indexes to the public key, and a second, “names” where the port number indexes to the name.

```
@app.route("/connect", methods=["POST"])
def accept_connection():
   global admin
   try:
       username = request.form.get("name")
       if username in names.values(): # if a user joins with an already
existing name, we append a number to differentiate
           n = 0
           for key in names.keys():
               if names[key] == username or (names[key][:len(names[key])-4]
== username):
                   n += 1
```

```
            username += " (" + str(n) + ")"

        user_port = request.form.get("port")
        key = request.form.get("public_key")

        if len(members) == 0:# if there are no members currently on the
server, the new member is made admin
            admin = user_port

        members.update({user_port: key})
        names.update({user_port: username})

        response_data = make_response("Account created successfully for " +
username + " on port " + str(user_port))
        response_data.status_code = 200

    except Exception as e:
        print(e)
        response_data = make_response(str(e))
        response_data.status_code = 400

    return response_data
```

Then in practice, whenever a client wishes to send a message, it runs the following method to get the public keys from the server,

```
def request_keys(port):
    body = {"origin": port} #included in the request so the server can easily
check if we should be allowed to access this data
    response = requests.post(("http://127.0.0.1:" + str(SERVER_PORT) +
"/key_request"), data=body)
    if response.status_code == 200:
        return response.json() #if the response is a success, return the
public keys to the send method
    else:
        print("Failed to acquire keys. error:\n", str(response.text), "\n\n")
```

Which calls this method on the server side.

```
@app.route("/key_request", methods=["POST"])
def return_keys():
    sender = request.form.get("origin")
    if sender not in members.keys(): # if we don't recognise the client, we
don't give them the keys
        response = make_response("unauthorised user")
        response.status_code = 403
        return response

    try:
        response = jsonify(members)
        response.status_code = 200
        response.headers["Content-Type"] = "application/json"

    except Exception as e:
```

```
        print(e)
        response = make_response(str(e))
        response.status_code = 400

    return response
```

If the client's port number is not recognised, we don't give them the keys. Otherwise, we just return them as normal.

This way, the sender can encrypt the message for each recipient before forwarding to the server. (kicking logic will be explained later)

```
def send(port):
    while True:
        message = input("")

        sys.stdout.write("\033[F")
        sys.stdout.write("\033[K")

        if message.startswith("|||"): #again we use ||| as a special
delimiter, this time to indicate a kick command
            kick(message.removeprefix("|||"), port)

        else: #without the ||| indicator, the typed content is sent as a
normal message
            keys = request_keys(port) #request the public keys of other
members from the server so we can encrypt the message for each
            body = {}
            for key in keys.keys(): #for each other member, encrypt the
message using their public key
                encoded = encrypt(message,
serialization.load_pem_public_key(keys[key].encode(),
backend=default_backend()))
                encoded = base64.b64encode(encoded).decode("utf-8")
                body.update({key: encoded})

            body.update({"origin": port}) #origin indicates sender


            response = requests.post(("http://127.0.0.1:" + str(SERVER_PORT) +
"/send"), data=body)

            if response.status_code != 200:
                print("Failed to send. error:\n", str(response.text), "\n\n")
```

And on the server side, we simply receive the message and forward each encrypted message to the intended recipient.

```
@app.route("/send", methods=["POST"])
def forward():
    sender = request.form.get("origin")
    if sender not in members.keys(): # if we don't recognise the client we
don't forward their message
```

```
        response = make_response("unauthorised user")
        response.status_code = 403
        return response

    try:
        sender = names[sender] + "|||" # we append our special delimiter to
the name so we can send the name and message as one packet and the clients
can separate them
        for member in members.keys():
            body = request.form.get(member)
            svr_skt.sendto((sender.encode("utf-8") + body.encode("utf-8")),
("127.0.0.1", int(member)))

        response = make_response("")
        response.status_code = 200

    except Exception as e:
        print(e)
        response = make_response(str(e))
        response.status_code = 400

    return response
```

Requirement 2 - Kicking Members:

The kicking of users from the group chat is also quite simple. First things first, this small snippet from the accept_connection method on the server side assigns whoever is the first user to join as the admin, uniquely allowing them to remove other users if they so choose.

```
if len(members) == 0: # if there are no members currently on the server, the
new member is made admin
    admin = user_port
```

This is the server side method that is called by clients that attempt to kick another user.

```
@app.route("/kick", methods=["POST"])
def kick():
    global admin
    sender = request.form.get("origin")
    if sender not in members.keys():
        response = make_response("unauthorised user")
        response.status_code = 403
        return response

    try:
        target = request.form.get("target")
        response = {}
        if sender == admin: # only the admin can disconnect other users
```

```
            if target == names[admin]: # if the admin is disconnecting
themselves, we assign the second oldest user as the new admin
                for key in members.keys():
                    admin = key
                    break
            if target in names.values(): # make sure the target actually
exists
                for key, value in names.items():
                    if value == target:
                        del members[key]
                        del names[key]
                        break
            else: # if the target doesn't exist
                response = make_response("that user does not exist")
                response.status_code = 404
        else: # if someone other than the admin attempts a kick
            if names[sender] == target: # if they want to disconnect
themselves
                for key, value in names.items():
                    if value == target:
                        del members[key]
                        del names[key]
                        break
            else: # if they want to disconnect someone else
                response = make_response("you do not have permission to
disconnect other users")
                response.status_code = 403

    except Exception as e:
        print(e)
        response = make_response(str(e))
        response.status_code = 400

    return response
```

As usual, if we don't recognise the client's port number we don't process the request. Otherwise, we have some checks in place to make sure it is the admin that is attempting to kick, and that the user they wish to kick does actually exist. However, we do also allow non-admin users to kick only themselves.
If the admin removes themselves from the group, we assign the next-oldest member of the group to be the new admin.

On the client side, the kicking logic is as below. The kick method is called by the threaded "send" method, whenever a message is prefixed by "|||". This acts as a special indicator for when a member wishes to kick. So, for example, if a member wished to kick a user named Peter, they'd simply send the message "|||Peter".

```
def kick(target, port):
    body = {"origin": port, "target": target} # origin identifies the sender
so the server can check permissions
```

```
    response = requests.post(("http://127.0.0.1:" + str(SERVER_PORT) +
"/kick"), data=body)

    if response.status_code == 200:
        print(target, " was kicked successfully\n")
    else:
        print("Failed to kick ", target, ". error:\n", str(response.text),
"\n\n")


def send(port):
    while True:
        message = input("")

        sys.stdout.write("\033[F")
        sys.stdout.write("\033[K")

        if message.startswith("|||"): #again we use ||| as a special
delimiter, this time to indicate a kick command
            kick(message.removeprefix("|||"), port)

        else: #without the ||| indicator, the typed content is sent as a
normal message
            keys = request_keys(port) #request the public keys of other
members from the server so we can encrypt the message for each
            body = {}
            for key in keys.keys(): #for each other member, encrypt the
message using their public key
                encoded = encrypt(message,
serialization.load_pem_public_key(keys[key].encode(),
backend=default_backend()))
                encoded = base64.b64encode(encoded).decode("utf-8")
                body.update({key: encoded})

            body.update({"origin": port}) #origin indicates sender


            response = requests.post(("http://127.0.0.1:" + str(SERVER_PORT) +
"/send"), data=body)

            if response.status_code != 200:
                print("Failed to send. error:\n", str(response.text), "\n\n")
```

Raw code below:

server.py-

```
import socket
from flask import Flask, request, jsonify, make_response
import threading
import time
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import rsa
```

```
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding

SERVER_PORT = 5000
members = {} # {port: public key}
names = {} # {port: name}
global admin
admin = None # tracks who is admin based on their port number

app = Flask(__name__)
svr_skt = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)
svr_skt.bind(("127.0.0.1", 5001)) # socket for forwarding messages

# called by clients that want to connect to the server
@app.route("/connect", methods=["POST"])
def accept_connection():
   global admin
   try:
       username = request.form.get("name")
       if username in names.values(): # if a user joins with an already
existing name, we append a number to differentiate
           n = 0
           for key in names.keys():
               if names[key] == username or (names[key][:len(names[key])-4]
== username):
                   n += 1
           username += " (" + str(n) + ")"

       user_port = request.form.get("port")
       key = request.form.get("public_key")

       if len(members) == 0: # if there are no members currently on the
server, the new member is made admin
           admin = user_port

       members.update({user_port: key})
       names.update({user_port: username})

       response_data = make_response("Account created successfully for " +
username + " on port " + str(user_port))
       response_data.status_code = 200

   except Exception as e:
       print(e)
       response_data = make_response(str(e))
       response_data.status_code = 400

   return response_data

#called by clients that want to access the other users' public keys
@app.route("/key_request", methods=["POST"])
def return_keys():
```

```
    sender = request.form.get("origin")
    if sender not in members.keys(): # if we don't recognise the client, we
don't give them the keys
        response = make_response("unauthorised user")
        response.status_code = 403
        return response

    try:
        response = jsonify(members)
        response.status_code = 200
        response.headers["Content-Type"] = "application/json"

    except Exception as e:
        print(e)
        response = make_response(str(e))
        response.status_code = 400

    return response

#called by clients that wish to send a message
@app.route("/send", methods=["POST"])
def forward():
    sender = request.form.get("origin")
    if sender not in members.keys(): # if we don't recognise the client we
don't forward their message
        response = make_response("unauthorised user")
        response.status_code = 403
        return response

    try:
        sender = names[sender] + "|||" # we append our special delimiter to
the name so we can send the name and message as one packet and the clients
can separate them
        for member in members.keys():
            body = request.form.get(member)
            svr_skt.sendto((sender.encode("utf-8") + body.encode("utf-8")),
("127.0.0.1", int(member)))

        response = make_response("")
        response.status_code = 200

    except Exception as e:
        print(e)
        response = make_response(str(e))
        response.status_code = 400

    return response

#called by clients that attempt to kick another user
@app.route("/kick", methods=["POST"])
def kick():
    global admin
    sender = request.form.get("origin")
```

```
    if sender not in members.keys():
        response = make_response("unauthorised user")
        response.status_code = 403
        return response

    try:
        target = request.form.get("target")
        response = {}
        if sender == admin: # only the admin can disconnect other users
            if target == names[admin]: # if the admin is disconnecting
themselves, we assign the second oldest user as the new admin
                for key in members.keys():
                    admin = key
                    break
            if target in names.values(): # make sure the target actually
exists
                for key, value in names.items():
                    if value == target:
                        del members[key]
                        del names[key]
                        break
            else: # if the target doesn't exist
                response = make_response("that user does not exist")
                response.status_code = 404
        else: # if someone other than the admin attempts a kick
            if names[sender] == target: # if they want to disconnect
themselves
                for key, value in names.items():
                    if value == target:
                        del members[key]
                        del names[key]
                        break
            else: # if they want to disconnect someone else
                response = make_response("you do not have permission to
disconnect other users")
                response.status_code = 403

    except Exception as e:
        print(e)
        response = make_response(str(e))
        response.status_code = 400

    return response


if __name__ == "__main__":
    app.config['PROPAGATE_EXCEPTIONS'] = True
    app.run(host="127.0.0.1", port=SERVER_PORT)
```

client.py

```
import socket
```

```
import requests
import threading
import sys
import base64
import time
from flask import Flask, request, jsonify, make_response
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding

#some constants
SERVER_PORT = 5000
BUFFER_SIZE = 4096

#method to connect to the server, runs automatically on start
def connect(key, name, port):
   key_bytes = key.public_bytes(encoding=serialization.Encoding.PEM,
format=serialization.PublicFormat.SubjectPublicKeyInfo)
   body = {"name": name, "port": port, "public_key": key_bytes} #name,
listening socket port and public key; this is everything the server needs to
manage each client
   while True:
       response = requests.post(("http://127.0.0.1:" + str(SERVER_PORT) +
"/connect"), data=body)

       if response.status_code == 200:
           print(response.text, "\n") #successful connection
           return
       else:
           print("Failed to connect. error:\n", str(response.text), "\n\n")
           time.sleep(5) #if we fail to connect, we wait 5 seconds before
trying to connect again

#this method gets called by the send messages method
def request_keys(port):
   body = {"origin": port} #included in the request so the server can easily
check if we should be allowed to access this data
   response = requests.post(("http://127.0.0.1:" + str(SERVER_PORT) +
"/key_request"), data=body)
   if response.status_code == 200:
       return response.json() #if the response is a success, return the
public keys to the send method
   else:
       print("Failed to acquire keys. error:\n", str(response.text), "\n\n")

#basic encyption function
def encrypt(message, key):
   return key.encrypt(message.encode(), padding.OAEP(
       mgf=padding.MGF1(algorithm=hashes.SHA256()),
algorithm=hashes.SHA256(), label=None))
```

```
#basic decryption
def decrypt(message, key):
   return key.decrypt(message, padding.OAEP(
           mgf=padding.MGF1(algorithm=hashes.SHA256()),
algorithm=hashes.SHA256(), label=None))

#method to listen for new messages, runs in a thread concurrent to the
sending method.
def listen(soc, key):
   while True:
       content = soc.recv(BUFFER_SIZE)
       name, message = content.split(b'|||') #"|||" is a special delimiter
for this system. in this case it separates the name of the sender and the
message
       name = name.decode("utf-8")
       message = decrypt(base64.b64decode(message), key)

       current_time_seconds = time.time()
       local_time = time.localtime(current_time_seconds)
       formatted_time = time.strftime("%H:%M:%S", local_time)

       print(name, formatted_time, "\n", message.decode(), "\n\n")
       #printed format:
       #sender HH:MM:SS
       #   message


def kick(target, port):
   body = {"origin": port, "target": target} # origin identifies the sender
so the server can check permissions
   response = requests.post(("http://127.0.0.1:" + str(SERVER_PORT) +
"/kick"), data=body)

   if response.status_code == 200:
       print(target, " was kicked successfully\n")
   else:
       print("Failed to kick ", target, ". error:\n", str(response.text),
"\n\n")


def send(port):
   while True:
       message = input("")

       sys.stdout.write("\033[F")
       sys.stdout.write("\033[K")

       if message.startswith("|||"): #again we use ||| as a special
delimiter, this time to indicate a kick command
           kick(message.removeprefix("|||"), port)

       else: #without the ||| indicator, the typed content is sent as a
normal message
```

```
            try:
                keys = request_keys(port) #request the public keys of other
members from the server so we can encrypt the message for each
                body = {}
                for key in keys.keys(): #for each other member, encrypt the
message using their public key
                    encoded = encrypt(message,
serialization.load_pem_public_key(keys[key].encode(),
backend=default_backend()))
                    encoded = base64.b64encode(encoded).decode("utf-8")
                    body.update({key: encoded})

                body.update({"origin": port}) #origin indicates sender


                response = requests.post(("http://127.0.0.1:" +
str(SERVER_PORT) + "/send"), data=body)

                if response.status_code != 200:
                    print("Failed to send. error:\n", str(response.text),
"\n\n")

            except Exception as e:
                print(e)


def main(argv):
    name = argv[0]
    port = int(argv[1])
    skt = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)
    skt.bind(("127.0.0.1", port)) #a socket with which to listen for incoming
messages

    private_key = rsa.generate_private_key(public_exponent=65537,
key_size=2048, backend=default_backend())
    public_key = private_key.public_key()

    connect(public_key, name, port) #register with the server

    send_thread = threading.Thread(target=send, args=(port,))
    send_thread.start() #start a thread that reads inputs and forwards
messages

    listen(skt, private_key)

if __name__ == "__main__":
    main(sys.argv[1:])
```