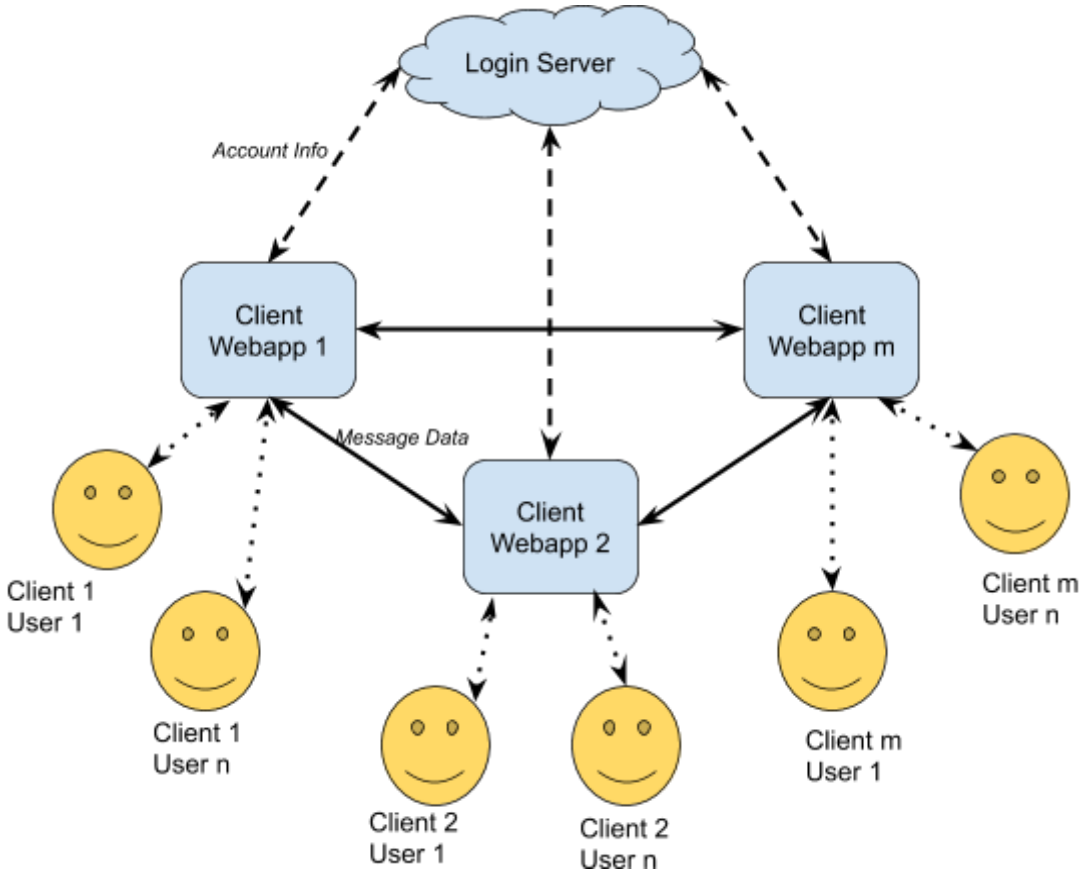
2019 COMPSYS 302 Class Protocol V8

Warning: This may not be the final specification for this document. Breaking changes may be introduced as the class implement the endpoints and find bugs.

INTRODUCTION

The COMPSYS 302 project involves the implementation of a hybrid peer-to-peer distributed system for a social media system. The general architecture is depicted here:



As can be seen, the overall architecture breaks communication for each client into three distinct types:

1) Data To/From the Login Server
2) Data To/From other Client Webapps
3) Data To/From User Connections

This document is structured around the first two connection types, i.e. 1) and 2).
This is because type 3) connections are entirely client-specific, and are made between a User web browser (e.g. Chrome) and the Client Webapp. They may be customised with no regard to the rest of the system.

GENERAL USAGE

SIGN IN FLOW

1. A user will enter a username and password into a client webapp
2. The client webapp will authenticate the username/password against the login server
    a. This can be done using /api/ping, or, better
    b. This can be done using /api/load_new_apikey
3. The client webapp will ensure that they have a public/private keypair
    a. it may choose to create a new keypair and publish to /api/add_pubkey
    b. It may choose to load existing keypair(s) (if any are available) from /api/get_privatedata
4. The client webapp should test the public/private keypair against the login server
    a. Using /api/ping
5. The client webapp should report the user is now available online and the connection info for the user
    a. Using /api/report
    b. You are not considered "signed in to the network" until you have called /api/report
6. The client webapp should now report to the user that they are online

SEND BROADCAST FLOW

1. A user will enter a message to broadcast
2. The webapp will sign their message
3. The webapp will send it by calling /api/rx_broadcast on all remote clients

SEND PRIVATE MESSAGE FLOW

1. A user will find another online user in their web app
    a. The webapp can do this by regularly polling /api/list_users
2. The user will enter a message to the destination user
3. The webapp will encrypt their message to the destination user
4. The webapp will send it by calling /api/rx_privatemessage on the remote client

SIGN OUT FLOW

1. Send a report to /api/report with status 'offline'
2. (If applicable) Delete the API key from memory

CONNECTION TYPE 1): CLIENT WEBAPPS TO LOGIN SERVER

In general, Users of the Client Webapps will authenticate themselves against a centralised 'Login Server'. This 'Login Server' is responsible for facilitating the discovery and authentication of each user on the entire network, and their *public* details will be stored on this remote server.

So that there may be trust within the network, users may choose to upload *public* keys to the login server. Then, Client Webapps may request these public keys for the purposes of authenticating messages sent from individual users.

The Login Server thus provides two main roles in this proposed architecture. Firstly, it serves as a mechanism for Client Webapps to discover one another on the internet, as they can register their connection information with it.
Secondly, it serves as a repository for *user information*, including for *public keys*. That way, when a digitally signed message is presented to a Client, it may verify the public key against the Login Server and ensure it matches a known username.

So that users may *migrate* between Client Webapps, an additional mechanism is provided for storing a symmetrically encrypted *private key* to the login server.

For the purposes of incoming messages, users may select which of their public keys should be used for encryption.

API DETAILS

The Login Server provides a set of APIs which may be communicated with via JSON over GET and POST requests.

APIs that require authentication require either

1) HTTP BASIC authentication with a valid username/password (recommended), OR
2) Headers “X-username” containing a valid Username and “X-apikey” containing a valid API key (useful for supplementary scripts)

SIGNATURE, PRIVATE KEY, AND PUBLIC KEY STRINGS

All public keys, private keys and signatures, across ALL API endpoints as well as via the ‘signature’ HTTP headers are taken from PyNaCl using 256-bit Ed25519 format with HexEncoded strings.

SIGNING

Some messages require signature fields. These are defined using the nomenclature “sign(...)”, where the computation methodology is defined using https://pynacl.readthedocs.io/en/stable/signing/ , and the bytes to sign are the arguments to “sign(...)”

E.g.

sign(a, b, c) = concatenate bytes of a, b, c, in that order, and pass to `.sign`

RESPONSE STATUS CODES

In general, the following HTTP Status Codes may be returned:
200: Okay
400: You have made a malformed or otherwise inappropriate request
401: You have not provided authorisation for a authenticated-required endpoint
404: You have accessed a non-existent endpoint
500: A malfunction has occurred within the server

Furthermore, the server will add a HTTP response header for X-*signature*, which will sign the returned messages for all endpoints.

Unless otherwise specified, all fields are compulsory. Each API is detailed on a single page.

✓ Login Server /api/ping

Method: POST
Requires Authentication: Optional
Purpose: Returns an “ok” message. Use to check if the login server is online and to test your signing/authentication

Send parameters:

Parameter Name	**Format**	**Purpose / Calculation**
pubkey (optional)	Public key string	A public key associated with your account
signature (optional)	Signature string	Sign(pubkey) (see Page 2)

Return parameters

Parameter Name	**Format**	**Purpose**
response	String	Says ‘ok’ if request ok, or ‘error’ if not
message (optional)	String	Contains error message (if applicable)
server_time	String of Float unix timestamp	Contains server time. Useful for finding out what time the login server thinks it is.
authentication	string	Says “basic” if the user has authenticated successfully with an appropriate username/password using basic, “api-key” if they are using an api-key, “error” if they haven’t, or “n/a” if no Authentication header is provided
signature	string	Says “ok” if the user has provided a valid public key associated with their account and successfully signed the message, “bad signature” if they haven’t provided a suitable signature, “bad pubkey” if they have provided an invalid pubkey or it isn’t associated with their account, or “n/a” if either the “oubkey” or “signature” fields are missing from posted JSON

Example:

POST /api/ping
Authorization: Basic …….

Body:

```
{
  "pubkey": "...",
  "signature": "..."
}
```

Response:

```
{
  "response": "ok",
  "server_time": "1556930832.3119302",
  "authentication": "basic",
  "signature": "ok"
}
```

✓ Login Server /api/list_apis

Method: GET
Requires Authentication: No
Purpose: Returns a list of APIs supported by this server.

Example:

GET /api/list_apis

Response:
(Excluded for length reasons)

✓ Login Server /api/load_new_apikey

Method: GET
Requires Authentication: True
Purpose: Returns a new API key for the purposes of authentication for the rest of this session. You use an API key in place of HTTP BASIC authentication. Provide an X-username header variable containing your username, and a X-apikey header variable containing the API key.

It is not possible to return the API key again, so make sure you save it (and discard when appropriate). Each time this endpoint is called a new API key will be generated.

The usage of API keys in general is optional - all auth-required endpoints will accept HTTP BASIC.

Return parameters

Parameter Name	**Format**	**Purpose**
response	String	Says ‘ok’ if request ok, or ‘error’ if not
message (optional)	String	Contains error message (if applicable)
api_key	string	Contains the API key for this session
api_key_generated_at	String of Float unix timestamp	Contains the time the API key was generated on the server.

Example:

GET /api/ping
Authorization: Basic …….

Response:

```
{
  "response": "ok",
  "api_key": " ……. ",
  "api_key_generated_at": "1556930832.3119302"
}
```

✓ Login Server /api/loginserver_pubkey

Method: GET
Requires Authentication: No
Purpose: Return the public key of the login server, which may be used to validate loginserver_records in broadcasts / privatemessages

Return parameters

Parameter Name	**Format**	**Purpose**
response	String	Says ‘ok’ if request ok, or ‘error’ if not
message (optional)	String	Contains error message (if applicable)
server_name	string	Says “cs302.kiwi.land”
server_time	String of Float unix timestamp	The time of the server
pubkey	Pubkey string	Contains the public key of the server

Furthermore, the server will add a HTTP response header for X-*signature*, which will sign the returned JSON bytes. (This is actually true of all messages received from the login server)

Example:

GET /api/loginserver_pubkey

Response:

```
{
"response": "ok",
"server_name": "cs302.kiwi.land",
"server_time": "1558328111.9488223",
"pubkey":
"21842e4bd1157058c61bc7d6edeb13b54c8215c849f6221c5c925b6c98cb47d1"
}
```

✓ Login Server /api/report

Method: POST
Requires Authentication: Yes
Purpose: Use this API to inform the login server about connection information for a user on a client. Transmit the details for this user at least once every five minutes and at most once every thirty seconds.

Send parameters:

Parameter Name	**Format**	**Purpose**
connection_address	String of ip:port	the public ip:port of this web client.
connection_location	Integer, either 0, 1, or 2	see the last section of this report.
incoming_pubkey	Public key string	the public key to encrypt messages against. The key must already be associated against the account (e.g. by adding it using /api/add_pubkey)
status (optional)	String, may be ‘online’, ‘away’, ‘busy’, ‘offline’	Informs other clients of your status. Send ‘offline’ as a final report to “sign out”. Defaults to ‘online’ if unspecified. Send a report with ‘offline’ to formally sign out of the network.

Return parameters:

Parameter Name	**Format**	**Purpose**
response	String	Says ‘ok’ if request ok, or ‘error’ if not
message (optional)	String	Contains error message (if applicable)

Example:

POST /api/report
Authorization: Basic …….

Body:

```
{
  "connection_address": "127.0.0.1:8000",
  "connection_location": 2,
  "incoming_pubkey": "...."
}
```

Response:


```
{
  "response": "ok"
}
```

✓ Login Server /api/list_users

Method: GET
Requires Authentication: Yes
Purpose: Use this API to load the connection details for all active users who have done a /report in the last five minutes to the login server.

Return Parameters:

Parameter Name	**Format**	**Purpose**
response	String	Says ‘ok’ if request ok, or ‘error’ if not
message (optional)	String	Contains error message (if applicable)
users	(array of next)	A list of users with the following fields
user.username	string	Client’s username
user.connection_address	ip:port	the public ip:port of this web client.
user.connection_location	Integer, either 0, 1, or 2	see the last section of this report
user.incoming_pubkey	Public key string	the public key to encrypt private messages addressed to this user against
user.connection_updated_at	String of Float unix timestamp	unix timestamp of login server during last report
user.status	String, may be ‘online’, ‘away’, ‘busy’, ‘offline’	That user’s self-reported status

Example:

GET /api/list_users
Authorization: Basic …….

Response:

```
{
  "users": [
    {
      "username": "admin",
      "connection_address": "127.0.0.1:8000",
      "connection_location": 2,
      "incoming_pubkey": " …….. ",
      "connection_updated_at": "1556930832.3119302",
```

```
      "status": "online"
    }
  ],
  "response": "ok"
}
```

✓ Login Server /api/add_pubkey

Method: POST
Requires Authentication: Yes
Purpose: Use this API to associate a public key (256-bit Ed25519 format, hex encoded) with your account. The public key that is added is the one provided for the purposes of the signature.

Send parameters:

Parameter Name	**Format**	**Purpose**
pubkey	Pubkey string	The public key we are adding to this user
username	string	The client's username (should match the username provided in the Authentication)
signature	Signature string	Sign(pubkey, username)

Return parameters:

Parameter Name	**Format**	**Purpose**
response	String	Says 'ok' if request ok, or 'error' if not
message (optional)	String	Contains error message (if applicable)
loginserver_record	Login Server Username/Pubkey Signature record Format is username,pubkey,server_time,signature	This string should be saved by the client to be used as a kind of 'certificate' to prove that the login server confirms this username/pubkey combination

The public key provided in the message should be derived from the private key used to sign the message (i.e. what is used to create X-signature HTTP header)

Example:

POST /api/add_pubkey
Authorization: Basic …….
X-signature: …….

Body:

```
{
  "pubkey": "...",
  "username": "admin",
  "signature": "......... "
}
```

Response:

```
{
  "loginserver_record": "admin,..........,1556930832.3119302, ......... "
}
```

✓ Login Server /api/get_loginserver_record

Method: GET
Requires Authentication: Yes
Purpose: Use this API to load your current loginserver_record for use in creating point-to-point messages. Change it by creating new pubkeys with /api/add_pubkey and/or by changing the current pubkey in /api/report.

Return parameters:

Parameter Name	**Format**	**Purpose**
response	String	Says ‘ok’ if request ok, or ‘error’ if not
message (optional)	String	Contains error message (if applicable)
loginserver_record	Login Server Username/Pubkey Signature record Format is username,pubkey, server_time,signature	This string should be saved by the client to be used as a kind of ‘certificate’ to prove that the login server confirms this username/pubkey combination

The public key provided in the message should be derived from the private key used to sign the message (i.e. what is used to create X-signature HTTP header)

Example:

GET /api/get_loginserver_record
Authorization: Basic …….

Response:

```
{
  "loginserver_record": "admin, ........ ,1556930832.3119302, ......... "
}
```

✓ Login Server /api/check_pubkey

Method: GET
Requires Authentication: Yes
Purpose: Use this API to load the loginserver_record for a given Ed25519 public key.
Send parameters:

Parameter Name	**Format**	**Purpose**
pubkey	string (256-bit Ed25519 format, Hex encoded)	The public key to determine the owner of

Return parameters:

Parameter Name	**Format**	**Purpose**
response	String	Says ‘ok’ if request ok, or ‘error’ if not
message (optional)	String	Contains error message (if applicable)
loginserver_record	Login Server Username/Pubkey Signature record Format is username,pubkey, server_time,signature	This string may be used as a kind of ‘certificate’ to prove that the login server confirms this username/pubkey combination
username	string	The username of the user owning this key
connection_address	String of ip:port	the public ip:port of their web client.
connection_location	Integer, either 0, 1, or 2	see the last section of this report.
incoming_pubkey	Public key string	the public key to encrypt messages against when sending messages to that user
connection_updated_at	String of Float unix time	The server time during the last /api/report by this user

Example:

GET /api/check_pubkey?pubkey= …..
Authorization: Basic …….

Response:


```
{
  "response": "ok",
  "loginserver_record": "admin,1556930832.3119302,1556930832.3119302, ......... ",
  "username": "admin",
  "connection_address": "127.0.0.1:8000",
  "connection_location": "2",
  "incoming_pubkey": " ……… ",
  "connection_updated_at": "1556930832.3119302",
}
```

✓ Login Server /api/add_privatedata

Method: POST
Requires Authentication: Yes
Purpose: Use this API to save symmetrically encrypted private data for a given user. It will automatically delete previously uploaded private data.
Send parameters:

Parameter Name	**Format**	**Purpose**
privatedata	String of secret data (Base64 encoded SecretBox)	A safe, encrypted storage blob of private data to be stored on the server. Cannot be larger than 4096 bytes. Other than length, content not examined/verified in any way on the server. *For detail on encryption and decryption of privatedata, see the end of the login server API section.*
loginserver_record	Login Server Username/Pubkey Signature record	This string may be used as a kind of 'certificate' to prove that the login server confirms this username/pubkey combination
client_saved_at	String of float unix timestamp	The time of the client when saving the data
signature	Signature string	sign(privatedata, loginserver_record, client_saved_at)

Private Data format (JSON dictionary, to be encrypted as privatedata)

Parameter Name	**Format**	**Purpose**
prikeys	list of private key strings	Lists all the private keys associated with this account
blocked_pubkeys	List of public key strings	Provides a way to store public keys that should be blocked by a client for their user

blocked_usernames	List of username strings	Provides a way to store usernames that should be blocked by a client for their user
blocked_message_signatures	List of message signature strings	Provides a way to store individual messages that should be blocked by a client for their user
blocked_words	List of words	Provides a way to store words to block on the client
favourite_message_signatures	List of message signature strings	Provides a way to store a record of individual messages that have been favorited by the user
friends_usernames	List of username strings	Provides a way to store usernames of your friends

For detail on encryption and decryption of privatedata, see the end of the login server API section.

Receive parameters:

Parameter Name	**Format**	**Purpose**
response	String	Says ‘ok’ if request ok, or ‘error’ if not
message (optional)	String	Contains error message (if applicable)
server_received_at	String of Float unix timestamp	The time of the server when saving the data

Example:

Private data is the following:

(Unencrypted privatedata):

```
{
  "prikeys": ["...", "..."],
  "blocked_pubkeys": ["...", "..."],
  "blocked_usernames": ["...", "..."],
  "blocked_words": ["...", "..."],
  "blocked_message_signatures": ["...", "..."],
  "favourite_message_signatures": ["...", "..."],
  "friends_usernames": ["...", "..."]
```

}

-------encryption--------> privatedata contents

POST /api/add_privatedata
Authorization: Basic …….
X-signature: …….

Body:

```
{
  "privatedata": "...",
  "pubkey": ".....",
  "client_saved_at": "1556930832.3119302",
  "signature": "..."
}
```

Response:

```
{
  "response": "ok",
  "server_received_at":  "1556930832.3119302"
}
```

✓Login Server /api/get_privatedata

Method: GET
Requires Authentication: Yes
Requires Signature: No
Purpose: Use this API to load the saved symmetrically encrypted private data for a user.
Receive parameters:

Parameter Name	**Format**	**Purpose**
response	String ‘ok’ or ‘error’ or ‘no privatedata available’	Informs whether or not data could be returned. If ‘no privatedata available’, then other fields will be omitted
message (optional)	String	Contains error message (if applicable)
privatedata (optional)	String of secret data (Base64 encoded SecretBox)	A safe, encrypted storage blob of private data to be retrieved from the server. *For detail on encryption and decryption of privatedata, see the end of the login server API section.*
loginserver_record (optional)	Login Server Username/Pubkey Signature record	Loginserver_record used to upload the data
client_saved_at (optional)	String of Float unix timestamp	The time of the client when data was saved
server_received_at (optional)	String of Float unix timestamp	The time of the server when data was received

Private Data format (See /api/add_privatedata)

Example:

GET /api/get_privatedata
Authorization: Basic …….

Response:
{

```
  "response": "ok",
  "privatedata": "...",
  "pubkey": ".....",
  "client_saved_at": "1556930832.3119302",
  "server_recieved_at": "1556930832.3119302"
}
```


-----deencryption---->

(Unencrypted privatedata):


```
{
  "prikeys": ["...", "..."],
  "blocked_pubkeys": ["...", "..."],
  "blocked_usernames": ["...", "..."],
  "blocked_words": ["...", "..."],
  "blocked_message_signatures": ["...", "..."],
  "favourite_message_signatures": ["...", "..."],
  "friends_usernames": ["...", "..."]
}
```

PRIVATEDATA ENCRYPTION AND DECRYPTION

It is suggested that the privatedata take the JSON form as described under the /api/load_privatedata endpoint.
Then, the instructions for encrypting it are as followed:

Steps for Secret_Box generation

1. Prompt the user for a new, unique password. This should not be the same as their loginserver log in password.
2. Using the nacl.pwhash.argon2i.kdf function, derive a symmetric key.
    a. This requires a key_password, which is the new password in byte form.
    b. salt: repeat the new password 16 times, then trim everything after 16 bytes.
    c. ops = nacl.pwhash.argon2i.OPSLIMIT_SENSITIVE
        i. # Recommended value of 8, given in the docs.
    d. mem = nacl.pwhash.argon2i.MEMLIMIT_SENSITIVE
        i. # Recommended value of 536870912, given in the docs.
3. Using the symmetric key, you may now create a secret_box.

Steps for Encrypting

1. Create a random nonce
    a. nonce = nacl.utils.random(nacl.secret.SecretBox.NONCE_SIZE) # Creates a random nonce.
2. Convert your object into a JSON formatted string, then convert it into bytes
3. Using the random nonce, create an encrypted_message using the secret_box from earlier.
4. Then, convert the encrypted_message into a base64 encoded string for transmission.

Steps for Decrypting

1. Convert the base64 encoded string into the raw bytes from earlier
2. Now, using a secret_box derived from the user's unique password, then you may decrypt the contents of the encrypted_message
3. Now you may convert the decoded bytes into a string, and decode as JSON

Useful links:
https://pynacl.readthedocs.io/en/stable/secret/

https://pynacl.readthedocs.io/en/stable/api/pwhash/

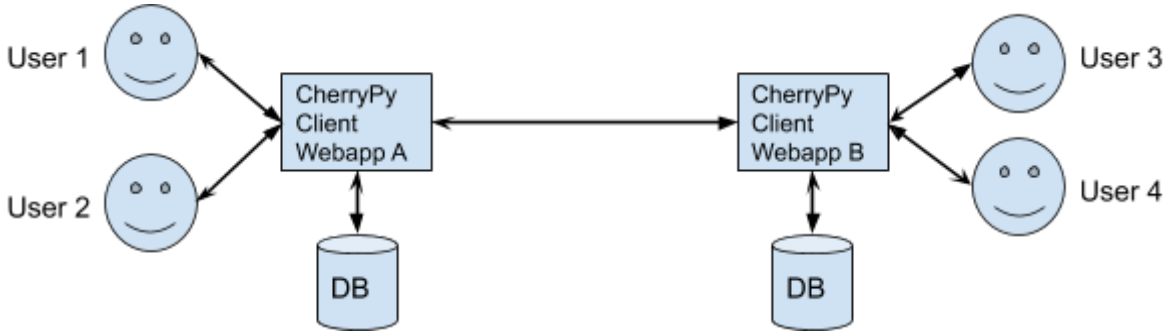
https://docs.python.org/2/library/base64.html

Note: As the marks for using privatedata correctly require another implementation to test against (i.e. can you save data on your implementation and then load them on another's), you may contact Jacob Allen (jall229@aucklanduni.ac.nz) who has got a working implementation.

CONNECTION TYPE 2): CLIENT WEBAPPS TO/FROM CLIENT WEBAPPS

Users of the Client Webapps will communicate with one another via messages to other Client Webapps. This process may be considered distinct from the Login/Authentication system, which is covered under Connection Type 1).

For the purposes of this system, refer to the following diagram, which provides a generalised architecture of the Client Webapp system.



There are multiple classifications of messages:

1) Public broadcast

This kind of message is when a user wants to make a “public statement” that will be transmitted to all users. It is the simplest to implement, as the message body is not encrypted. Messages are transmitted between **every** Client Webapp.
E.g. In the figure, User 1 makes a broadcast message which will be available to User 2, User 3, and User 4. As a result the broadcast must be transmitted between Webapp A and Webapp B.

2) Point-to-point message

This kind of message is from a single user to another single user.
E.g., In the figure, User 1 makes a private point-to-point message to User 3. As a result, the message must be transmitted between Webapp A and Webapp B. Users may also choose to transmit their messages to other Client Webapps.
E.g. In the figure, user 1 makes a private point-to-point message to User 2. Webapp A may realise that User 2 is currently co-located with User 1, so it makes the message available locally to User 2.

3) Offline point-to-point message

This kind of message is from a single user to another user, but the target user is offline. The sender may simply choose to wait until the receiver comes online, but this may not be practical (i.e. the sender might go offline first). Instead, the User may choose to transmit their

message to other nodes in the network. When properly encrypted, the other nodes, though they have copies of the data, will not be able to decrypt it.
Then, when the receiver comes online and asks other nodes if they have any messages for it, the other nodes should transmit the private message.

4) Group chat

This kind of message is a hybrid between 1) and 2). It is when there is a public broadcast of messages, but the message body is encrypted symmetrically. Then, specific users may have the secret key to decrypt the message body. They may be sent the keys via the endpoint /api/rx_groupinvite, as detailed below.
E.g. In the figure, User 1 creates a group chat for User 1, User 2, and User 4. He uses the point-to-point mechanism to transmit a secret symmetric key to Users 2 and 4. Then, messages are transmitted via the Broadcast mechanism.

To facilitate clients that may go offline and return online, each server is required to keep a copy of all public broadcast messages and should provide these on request.

In general, it is assumed that messages are plain text. However, you could also implement (in your client) the ability to display messages as *markdown*.

META MESSAGES

To facilitate further advanced functionality on top of the existing API, we define *meta messages*, which are special message strings encoded into the existing *broadcast* and *privatemessage* framework.

Meta messages are specially formatted strings which begin with "!Meta:". Then, further options can be transmitted, as per the table below:

Meta message format	Explanation
!Meta:favourite_broadcast:[broadcast_sig]	Replace [broadcast_sig] with an appropriate broadcast signature and send this as either a public broadcast or a privatemessage to indicate that you are favouriting a broadcast
!Meta:block_broadcast:[broadcast_sig]	Replace [broadcast_sig] with an appropriate broadcast signature and send this as either a public broadcast or a privatemessage to indicate that you are blocking a broadcast
!Meta:block_username:[username]	Replace [username] with an appropriate username and send this as either a public broadcast or a privatemessage to indicate that you are blocking a username
!Meta:block_pubkey:[pubkey]	Replace [pubkey] with an appropriate pubkey and send this as either a public broadcast or a privatemessage to indicate that you are blocking a public key

Can you think of additional meta messages? Let Hammond know, and he will add them to the specification!

✓ /api/rx_broadcast

Method: POST
Purpose: Use this API to transmit a signed broadcast between users. You need to be authenticated, and the broadcast public key must be known, but the broadcast public key need not be associated to your account.

Note: The login server also implements this, and for testing purposes messages can be sent to be viewed at cs302.kiwi.land

Send parameters

Parameter Name	**Format**	**Purpose**
loginserver_record	Login Server Username/Pubkey Signature record Format is username,pubkey, server_time,signature	This string may be used as a kind of 'certificate' to prove that the login server confirms this username/pubkey combination. It contains the pubkey and username of the poster of this message Server_time refers to the creation time of the loginserver_record
message	String	The message to be transmitted. Utf-8 string. Cannot exceed 256 bytes in length. Message is not encrypted.
sender_created_at	String of Float unix time	the local time of the user when this system was created
signature	Signature string	Sign(loginserver_record, message, sender_created_at)

Return parameters:

Parameter Name	**Format**	**Purpose**
response	String	Says 'ok' if request ok, or 'error' if not
message (optional)	String	Contains error message (if applicable)

Example:

POST /api/rx_broadcast

Body:

```
{
  "loginserver_record": " …… ",
  "message": "Hello world!",
  "sender_created_at" : "1556931977.0179243",
  "signature" : " ……."
}
```

Response:

```
{
  "response": "ok"
}
```

✓ /api/rx_privatemessage

Method: POST
Purpose: Use this API to transmit a secret message between users. Certain "meta"-information is public (the sender username/pubkey, and the destination username/pubkey, the timestamp), but the message itself is private.

Note: For testing purposes, you may call /api/rx_privatemessage on the login server with 'admin' as the target_username and the server's public key as the target_pubkey. Received messages will be posted to /messages_to_admin

Send parameters

Parameter Name	**Format**	**Purpose**
loginserver_record	Login Server Username/Pubkey Signature record Format is username,pubkey, server_time,signature	This string may be used as a kind of 'certificate' to prove that the login server confirms this username/pubkey combination. It contains the pubkey and username of the poster of this message Server_time refers to the creation time of the loginserver_record
target_pubkey	Pubkey string	The public key of the targeted user. This is what the message has been encrypted against.
target_username	String	The username of the targeted user
encrypted_message	String	The message to be transmitted. Utf-8 string. Cannot exceed 1024 bytes in length. Message is encrypted against the target_pubkey using the nacl *SealedBox*: see https://pynacl.readthedocs.io/en/stable/public/ A further implementation hint is below
sender_created_at	String of float unix time	the local time of the user when this system was created

signature	Signature string	Sign(loginserver_record, target_pubkey, target_username, encrypted_message, sender_created_at)

Return parameters:

Parameter Name	**Format**	**Purpose**
response	String	Says ‘ok’ if request ok, or ‘error’ if not
message (optional)	String	Contains error message (if applicable)

Example:

POST /api/rx_broadcast
Body:

```
{
  "loginserver_record": " …… ",
  "target_pubkey": " …… ",
  "target_username": " admin ",
  "encrypted_message": " … ... …."
  "sender_created_at" : "1556931977.0179243",
  "signature" : " ……."
}
```

Response:

```
{
  "response": "ok"
}
```

Implementation Hint:
When coding this you may discover that *signing* and *verifying* messages in PyNaCl uses Ed25519 type keys, but *encrypting* and *decrypting* messages uses Curve25519 type keys. These are not the same!
However, it is a straightforward process to get from an Ed25519 type key to a Curve25519 type key.

Hence, when constructing your *encrypted_message*, you should first convert the target_pubkey into a public nacl.signing.VerifyKey as normal.
Then, you can convert this to the appropriate type by calling .to_curve25519_public_key()

This will return the curve-type public key for using in the SealedBox functions.

Likewise, for decrypting, you will need to load your private SigningKey from your save data, then use the .to_curve25519_private_key() to convert this for use in the SealedBox functions.

Sample code (for encrypting):

```
#publickey_hex contains the target publickey
#using the nacl.encoding.HexEncoder format
verifykey = nacl.signing.VerifyKey(publickey_hex, encoder=nacl.encoding.HexEncoder)
publickey = verifykey.to_curve25519_public_key()
sealed_box = nacl.public.SealedBox(publickey)
encrypted = sealed_box.encrypt(message, encoder=nacl.encoding.HexEncoder)
message = encrypted.decode('utf-8')
print(message)
```

/api/checkmessages

Method: GET
Purpose: Use this API to retrieve already-sent messages from other clients in the network.

Note: The login server does not implement this endpoint.

Send parameters:

Parameter Name	**Format**	**Purpose / Calculation**
since	Float unix timestamp	The time since you were last online

Receive parameters:

Parameter Name	**Format**	**Purpose**
response	String	Says ‘ok’ if request ok, or ‘error’ if not
message (optional)	String	Contains error message (if applicable)
broadcasts	List of broadcasts in the format described in /api/rx_broadcast	Contains all broadcasts received by this server since time “since” (i.e. since you were last online)
private_messages	List of private messages in the format described in /api/rx_privatemessage	Contains all private messages received by this server since time “since” (i.e. since you were last online) These private messages may be from this or other servers, and may or may not be addressed to you.

Example:

GET /api/checkmessages?since=...

Response:

```
{
  "response": "ok",
  "broadcasts": [...],
  "private_messages": [...]
}
```

/api/ping_check

Method: POST
Purpose: Use this API to check if another client is active. Optionally you may report which users are currently using your client, and they may return the users on their client.

Send parameters

Parameter Name	**Format**	**Purpose / Calculation**
my_time	Float unix timestamp	Your client's current time
my_active_usernames (optional)	List of username strings	Contain a list of users currently active on your client
connection_address	String of ip:port	the public ip:port of your web client.
connection_location	Integer, either 0, 1, or 2	see the last section of this report.

Receive parameters:

Parameter Name	**Format**	**Purpose**
response	String	Says 'ok' if request and server is ok, or 'error' if not
message (optional)	String	Contains error message (if applicable)
my_time	Float unix timestamp	This client's current time
my_active_usernames (optional)	List of username strings	Contain a list of users currently active on this client

Example:

POST /api/rx_broadcast
Body:

```
{
  "my_time": "1556931977.0179243",
  "my_active_usernames": ["...", "...", "..."],
  "connection_address": "123.456.789.123",
  "connection_location": 2
}
```

Response:

```
{
```

```
  "response": "ok",
  "my_time": "1556931978.0179243",
  "my_active_usernames": ["...", "...", "..."]
}
```

/api/rx_groupmessage

Method: POST
Purpose: Use this API to transmit a secret group message between users. You need to be authenticated, and the broadcast public key must be known, but the secret group need not be associated to your account.

Note: The login server does not implement this endpoint.

Send parameters

Parameter Name	**Format**	**Purpose**
loginserver_record	Login Server Username/Pubkey Signature record Format is username,pubkey, server_time,signature	This string may be used as a kind of 'certificate' to prove that the login server confirms this username/pubkey combination. It contains the pubkey and username of the poster of this message Server_time refers to the creation time of the loginserver_record
groupkey_hash	String	This string may be used as a unique identifier for groups. It is defined as the sha256 hash of the symmetric key used to decode the group_message.
group_message	Encrypted String	The symmetrically encrypted group message to be transmitted. Utf-8 string. Cannot exceed 1024 bytes in length.
sender_created_at	String of Float unix time	the local time of the user when this system was created
signature	Signature string	Sign(loginserver_record, group_message, sender_created_at)

Return parameters:

Parameter Name	**Format**	**Purpose**
response	String	Says 'ok' if request ok, or 'error' if not
message (optional)	String	Contains error message (if applicable)

Example:

POST /api/rx_groupmessage

Body:

```
{
  "loginserver_record": " …… ",
  "groupkey_hash": " ……..... ",
  "group_message": " …… ",
  "sender_created_at" : "1556931977.0179243",
  "signature" : " ……."
}
```

Response:

```
{
  "response": "ok"
}
```

/api/rx_groupinvite

Method: POST
Purpose: Use this API to transmit a secret group message between users. You need to be authenticated, and the broadcast public key must be known, but the secret group need not be associated to your account.

Note: The login server does not implement this endpoint.

Send parameters

Parameter Name	**Format**	**Purpose**
loginserver_record	Login Server Username/Pubkey Signature record Format is username,pubkey, server_time,signature	This string may be used as a kind of 'certificate' to prove that the login server confirms this username/pubkey combination. It contains the pubkey and username of the poster of this message Server_time refers to the creation time of the loginserver_record
groupkey_hash	String	This string may be used as a unique identifier for groups. It is defined as the sha256 hash of the symmetric key used to decode the group_message.
target_pubkey	Pubkey string	The public key of the targeted user. This is what the message has been encrypted against.
target_username	String	The username of the targeted user
encrypted_groupkey	String	The key to symmetrically encode/decode group messages. The key is encrypted against the target_pubkey using the nacl SealedBox.
sender_created_at	String of Float unix time	the local time of the user when this system was created
signature	Signature string	Sign(loginserver_record, groupkey_hash, target_pubkey, target_username, encrypted_groupkey, sender_created_at)

Return parameters:

Parameter Name	**Format**	**Purpose**
response	String	Says ‘ok’ if request ok, or ‘error’ if not
message (optional)	String	Contains error message (if applicable)

Example:

POST /api/rx_groupinvite

Body:

```
{
  "loginserver_record": " …… ",
  "groupkey_hash": " ……..... ",
  "target_pubkey": " ……..... ",
  "target_username": " ……..... ",
  "encrypted_groupkey": " …… ",
  "sender_created_at" : "1556931977.0179243",
  "signature" : " ……."
}
```

Response:

```
{
  "response": "ok"
}
```

CONNECTION LOCATIONS

Because of the University's firewall and network configurations, there are various issues with certain IP addresses being able to connect to other IP addresses depending on where clients are located.

Requester / *Host*	0 – University Lab Desktop	1 – University Wireless	2 – Rest of the World
0 – University Lab Desktop	Valid (local)	Invalid	Invalid
1 – University Wireless	Invalid	Valid (local)	Invalid
2 – Rest of the World	Valid	Valid	Valid

This assumes a computer in the "rest of the world" has its ports forwarded correctly.

The location argument thus states which type of location the user is currently in. This is because with the issues between external and local IP addresses on the network, it is always not possible for the server to automatically determine your local IP address. However, clients may check that your reported IP address is within the expected range.

Use the following values:
0 – University Lab Desktop (local IP beginning with 10.103…)
1 – University Wireless Network (local IP)
2 – Rest of the world (external IP)