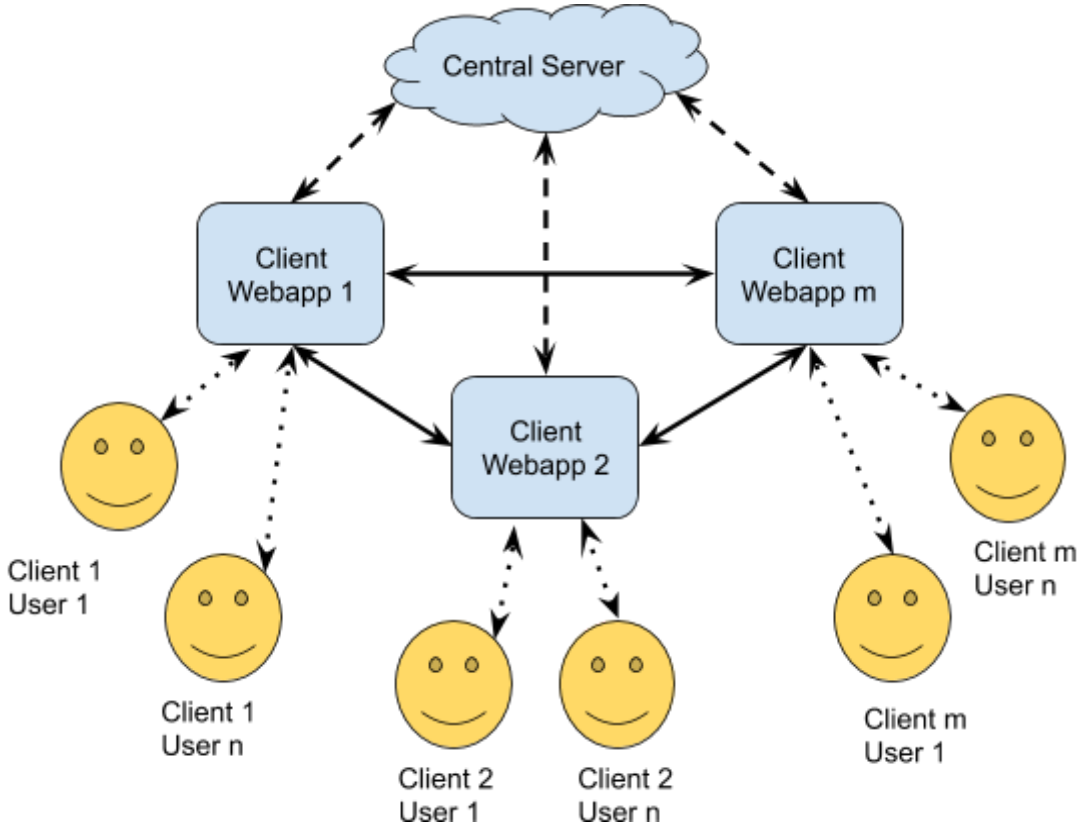
2019 COMPSYS 302 Login Server API

This is not a final document. This is the initial specification for the login server provided for the CS302 distributed social media (Python) project.

INTRODUCTION

The **login server** provides an example mechanism for distributed communication between web clients in a social media application. It represents the "central server" in a *hybrid* communication environment.

This can be visualised using the Figure below:



Here, Users of the Client Webapps will authenticate themselves against the Central Server, and their details will be stored on this remote server. Users can send broadcast messages amongst themselves, and to the central server. So that the messages can be trusted, they will be *digitally signed.* This means that for each User, there may be one (or more) set(s) of *public/private key pairs*.

The Central 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 can serve as a *repository* for *public keys*. That way, when a digitally signed message is presented to a Client, it may verify the public key against the Central Server and ensure it matches a known username.

USAGE

The Login Server / Central Server provides a set of APIs which will be extended as the project evolves. APIs communicate using JSON over GET and POST requests. APIs that require authentication require HTTP BASIC authentication with a valid username/password. To begin with, this is what is available.

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

All fields are compulsory. Use the Examples as

/api/ping

Method: GET
Requires Authentication: No
Purpose: Returns an “ok” message. Use to check if the login server is online.

Example:

GET /api/ping
Response:

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

/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)

/api/report
Method: POST
Requires Authentication: Yes
Purpose: Use this API to authenticate a client with the remote login server. Transmit the details for this user at least once every five minutes and at most once every thirty seconds.

Send parameters:	connection_location	see the last section of this report.
	connection_address	the public ip:port of this web client

Example:

POST /api/report
Body:

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

Response:

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

/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:	users	a list of users, with the following fields
	username	client's username
	connection_address	client's self-reported ip:port
	connection_location	see the last section of this report
	connection_updated_at	unix timestamp of login server during last /report

Example:

GET /api/list_users
Response:

```
{
  "users": [
    {
      "username": "admin",
      "connection_address": "127.0.0.1:8000",
      "connection_location": "2",
      "connection_updated_at": 1556930832.3119302
    }
  ],
  "response": "ok"
}
```

/api/add_pubkey
Method: POST
Requires Authentication: Yes
Purpose: Use this API to send a public key (256-bit Ed25519 format, Hex encoded) and associate it with your account.

Send parameters:	pubkey	the public key the user is adding.
	username	the username of the user
	signature	a signature to prove authenticity

Signature calculation: In this order, concatenate the bytes for the public key and the username - then, use the result as the message to be signed using the PyNaCl signature calculation as on https://pynacl.readthedocs.io/en/stable/signing/
The private key used to sign should be derived from the public key used in the message.

Example:

POST /api/add_pubkey
Body:

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

Response:

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

/api/check_pubkey
Method: GET
Requires Authentication: Yes
Purpose: Use this API to load the username for a given broadcast public key.

Send parameters:	pubkey	the public key (256-bit Ed25519 format, Hex encoded)

Example:

GET /api/check_pubkey?pubkey= .....
Response:

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

/api/rx_broadcast
Method: POST
Requires Authentication: Yes
Purpose: Use this API to publish a signed broadcast to the login server. 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.

Send parameters:	pubkey	the public key of the broadcast.
	message	the message to broadcast
	sender_created_at	the client's timestamp for the message
	signature	a signature to prove authenticity

Signature calculation: In this order, concatenate the bytes for the public key, the message, and the sender_created_at timestamp - then, use the result as the message to be signed using the PyNaCl signature calculation as on https://pynacl.readthedocs.io/en/stable/signing/ The private key used to sign should be derived from the public key used in the message.

Example:

POST /api/rx_broadcast
Body:

```
{
  "pubkey": " …… ",
  "message": "Hello world!",
  "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)