COMPSYS302 – Client Application APIs

This is not a final document – more APIs and/or details may be added as additional functionality is added. This document describes the API functionality of applications to be developed by the students. It allows developers to create applications that interact with each other programmatically without human interaction. Importantly, it only covers **inter-app** communication, as any intra-app or (within-app) implementation is left up to the developers. It also only covers **inbound** communication, as how the client initiates outbound communication is left up to the developers. This protocol document does not cover interaction with the login server, which is described in a separate document. All APIs and parameters are **case sensitive**. All communication is **HTTP only**.

JSON Encoding

Unless stated otherwise, all communication between nodes must be JSON encoded, even if there is only one argument. There is an important exception: **if the return is only an error code, it should not be JSON encoded**. This allows for serialisation of data/arguments, and provides a more flexible format for sending and receiving data. Below is one example of how to encode data into JSON:

```
      import json
      output_dict = {  "sender": "abcd001",
                 "message": "This is a test.",
                 "recipient": "wxyz999"}
      data = json.dumps(output_dict) #data is a JSON object
      return data
```

To decode data from JSON:

```
      import json
      input_dict = json.loads(input_data)
```

For all APIs expecting a JSON object, you should use urllib2.Request to encode the JSON object before sending it to the other client:

```
      import urllib2
      req = urllib2.Request(<URL>, <JSON Object>, {'Content-Type':
'application/json'})
      response = urllib2.urlopen(req)
```

To receive a JSON object from the API, CherryPy does the decoding for you automatically:

```
      @cherrypy.tools.json_in()
      def receiveData(self):
            input_data = cherrypy.request.json
```

Where input_data is the decoded dictionary. It is strongly suggested that you do more research into JSON online, and discuss how to use it with other students so that your implementation does not exist in isolation.

/ping (compulsory) [POST]

Allows another client to check if this client is still here before initiating other communication. This also allows someone to check that a valid application is running at this address:port combination. It can also allow a client to test the round-trip delay time for messages to another node. No JSON encoding is required for this API.

Parameters:	sender (username string, **required**)
Returns:	0 (representing error code 0, this API should always be successful)

/receiveMessage (compulsory) [POST]
This API allows another client to send this client a message. A single message, its metadata, and any control arguments should be in a dictionary and then JSON encoded. All messages sent and received should be stored locally. A unique message ID should be generated locally when receiving. Each client should do their own duplicate checking.

Parameters:	sender (username string, **required**)
	destination (username string, **required**)
	message (string, **required**)
	stamp (seconds since epoch time float, **required**)
	encoding (integer (see below), optional, for special characters)
	encryption (integer (see below), optional, for data security)
	hashing (integer (see below), optional, for integrity checking)
	hash (hexstring of hashed message (see below), optional, for integrity checking)
	decryptionKey (see encryption section below, optional)
	groupID (string, optional (only if relevant))
Returns:	<numerical error code>

/receiveFile (compulsory) [POST]
This API allows another client to send this client an arbitrary file (in binary). A single base64 encoded file, its metadata, and any control arguments should be in a dictionary and then JSON encoded. Files do not necessarily need to be stored locally when sending, but should be stored when receiving. A maximum file size of 5MB should be enforced for the purposes of prototype testing.

Parameters:	sender (username string, **required**)
	destination (username string, **required**)
	file (base64 sequence, **required**)
	filename (string including the extension, **required**)
	content_type (mimetype string, **required**)
	stamp (seconds since epoch time float, **required**)
	encryption (integer (see below), optional, for data security)
	hashing (integer (see below), optional, for integrity checking)
	hash (hexstring (see below), optional, for integrity checking)
	decryptionKey (see encryption section below, optional)
	groupID (string, optional (only if relevant))
Returns:	<numerical error code>

/acknowledge (optional) [POST]
If a client receiving a message decides that returning a numerical error code is not enough, then they can call a client's acknowledge API to reconfirm that the message or file was received correctly. By sending the hash of the message or file, the original sender can compare it against the hash of the original message or file, and affirm it was sent correctly. This should be done when the message or file is actually seen by the user in the browser, not when the message was received by the node.

Parameters:	sender (username string, **required**)
	stamp (seconds since epoch time float of the original message, **required**)
	hashing (integer (see below), **required**)
	hash (hexstring of hashed message (see below), **required**)
Returns:	<numerical error code>

/getPublicKey (optional) [GET]
If a client doesn't trust the public key that is currently reported onto the login server, then it should directly query this client for the public key for the current user/client pair.

Parameters:	sender (username string, **required**)
	username (username that you are requesting, string, **required**)
Returns:	error (numerical error code (see below), **required**)
	pubkey (hexstring of public key, **required**)

/handshake (optional) [POST]
This API allows another client to confirm that a particular encryption standard is supported correctly by this client. An encrypted message sent to this client should be decrypted, and then returned in plaintext for verification at the other end. This can also be used for verifying users/clients with their public keys as necessary.

Parameters:	message (string, **required**)
	sender (string, **required**)
	destination (string, **required**)
	encryption (integer (see below), **required**)
	decryptionKey (see encryption section below, required if encryption=4)
Returns:	error (numerical error code (see below), **required**)
	message (decrypted string, **required**)

/getProfile (compulsory) [GET]
This API allows another client to request information about the user operating this client. It implies that the current user has a profile existing on this client that contains the necessary information.

Parameters:	profile_username (username that you are requesting, string, **required**)
	sender (string, **required**)
Returns:	lastUpdated (seconds since epoch time when profile was fetched, float, **required)**
	fullname (the actual name of the user, string, optional)
	position (user's job title, string, optional)
	description (brief information about the user, string, optional)
	location (current location of the user, string, optional)
	picture (a URL for a jpg or png profile picture for the user, string, optional)
	encoding (integer (see below), optional, for special characters)
	encryption (integer (see below), optional, for data security)
	decryptionKey (see encryption section below, optional)

Group Messages:
See the login server API, which co-ordinates the formation and retrieval of groups. It is then up to the individual nodes to send and receive information based on the group, making sure to include the relevant GroupID so that recipients know that they are part of a group.

Login Server Outage:
In the event of a login server outage, the network could potentially still function by falling back to peer to peer networking methods. This should not be considered as secure as the login server.

/getList (optional) [GET]
Identical output as server /getList, maintaining the plaintext output. Essentially provides the current user list maintained by this client.

Parameters:	username (username string, **required**)
	json_format (1 if you prefer a JSON list instead of plain text, optional)
Returns:	<numerical error code>
	If the Error Code is 0, it is followed by a comma separated list of users:
	<username>,<location>,<ip>,<port>,<last login in epoch time>,<publickey(opt)>
	Epoch time is the number of seconds since January 1, 1970 GMT
	If json=1, a JSON dictionary is returned instead with
	the keys: username, location, ip, port, lastLogin, publicKey

/report (optional) [POST]
This API allows another client to add themselves to the local user list on this client. Input is expected to be JSON. The client calling this method should call this on all users returned by the above /getList to ensure that every client on the network maintains an up to date list of users. Clients should process a given signature using the last reported public key on the login server as described below, and verify that the hashes match – if so, then the user can be deemed trustworthy. All other clients should be considered untrustworthy by default and marked as offline. No certificates are used, which is less secure, but sufficient for our prototyping purposes. Clients should report at least once per minute, but not much more than that – rate limiting may be appropriate.

Parameters:	username
	passphrase (arbitrary string of at least 1024 characters)
	signature (username, encrypted with private key of report-ing client)
	location (integer, see login server protocol)
	ip
	port
	encryption (integer (see below), optional)
Returns:	<numerical error code>

/logoff (optional) [POST]
Clients should use this API to tell other nodes that they are leaving the network (without waiting for the timeout). The client must provide valid credentials to log off (to avoid false logoff requests).

Parameters:	username
	passphrase (arbitrary string of at least 1024 characters)
	signature (username, encrypted with private key of report-ing client)
	encryption (integer (see below), optional)
Returns:	<numerical error code>

At this stage, no API is available for:
- Global Messageboard
- Delete/Clear messages on both sender and receiver side
- Offline Messaging (without login server)

Developers wishing to implement this functionality should write draft API in conjunction with at least two other students, and submit them to the manager (Andrew).

Encoding:
By default, ASCII is assumed. For the relevant API, there is an optional **encoding** argument for any text that is to be displayed to the user. If there is an **encoding** argument, the following encoding standards are used:

0 – ASCII
1 – UTF-8
2 – UTF-16
3 – Unicode 10.0

Encryption:
By default, no encryption is assumed. For any set of data, there is an optional **encryption** argument. If there is an **encryption** argument, the following encryption standards are used:

0 – No Encryption
1 – XOR Encryption (key = "10010110")
2 – AES-256 Encryption (key = "41fb5b5ae4d57c5ee528adb078ac3b2e", block size 16 padded with spaces, CBC mode)
3 – RSA-1024 Encryption (locally generated key pair, public key on login server, DER mode, content must be ASCII strings only, message size less than 128 characters)
4 – RSA-1024 and AES-256 Encryption (with the decryptionKey argument, see below)
5 – RSA-2048 and AES-256 Encryption (with the decryptionKey argument, see below)

If there is an **encryption** argument (and the value is larger than 0), it is assumed that **all other arguments, except for destinations, senders, and encoding/encryption/hashing arguments,** in the API call are encrypted with that standard. If a client receives a message that is encrypted by a standard that it does not support, it should return the relevant error message (Err 9). All encrypted outputs must be **hexstrings**.

If a decryptionKey argument is provided for the relevant APIs, then the node should use their RSA-1024 private key to decrypt the value of that argument, and then use the resultant AES-256 key to decrypt the actual message itself. The sender of a message should generate an AES-256 key, encrypt the message using that key, and encrypt the key itself with the recipient's public key as found on the login server.

Hashing:
By default, no hashing is assumed. For any set of data, there is an optional **hashing** argument. If there is a **hashing** argument, the following hashing standards are used:

0 – No Hashing
1 – SHA-256 (with no salt)
2 – SHA-256 (message/file concatenated with sender username (in ASCII))
3 – SHA-512 (message/file concatenated with sender username (in ASCII))
4 – bcrypt (message/file concatenated with sender username (in ASCII))
5 – scrypt (message/file concatenated with sender username (in ASCII))

If a client does not support hashing of messages, then it can usually ignore the hash. If the client does support hashing, but receives a hash in a standard that it does not support, then it should return the relevant error message (Err 10).

Error codes and messages:
0: <Action was successful>
1: Missing Compulsory Field
2: Unauthenticated User
3: Client Currently Unavailable
4: Database Error
5: Timeout Error
6: Insufficient Permissions
7: Hash does not match
8: Encoding Not Supported
9: Encryption Standard Not Supported
10: Hashing Standard Not Supported
11: Rate Limited

Note that not all of these error codes/messages may be used, and more may be added.

External packages:
If you use an external package or library, you **must include it locally** in your submission. This includes both Python and Javascript libraries, and we would rather you didn't use npm. All javascript libraries are approved by default, although you are reminded that the focus of this project is on the Python backend, and the frontend design is just something that allows us to test the backend functionality.

Approved External Packages (Libraries):
CherryPy 3.7, PyCrypto 2.6.1, pytest 3.5.1, Requests 2.18.4, Bleach 2.1.3, Jinja2
and their dependencies

If you want to use any other external package/module/library, you must get permission from your manager first. To be clear, you are allowed to use any library that is in the standard distribution of Python – it's just ones that you have to download and install separately that have a limitation.