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. **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.

/listAPI (compulsory)

Lists the APIs supported by the client, as well as the encoding, encryption, and hashing standards. If the standards are not included, it is assumed that the client does not support any encoding (beyond ASCII), encryption, or hashing. The output is returned in plaintext (i.e. not JSON encoded).

Parameters: None

Returns: List of APIs in the format /<API> [<argument_name>] [<argument_name(opt)>]
Encoding <numbers representing supported standards with space between each>
Encryption <numbers representing supported standards with space between each >
Hashing <numbers representing supported standards with space between each >

/ping (compulsory)
Allows another client to check if this client is still here before initiating other communication. 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)
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. The same function is used for sending intermediaries a message for someone else who is currently offline (by setting the correct "destination"). If your node is kind, it will periodically push messages for people who are currently offline to other nodes as well. 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, optional)
markdown (1 if the message is in markdown format, otherwise 0)
encoding (integer (see below), optional, for extended character support)
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)
Returns: <numerical error code>

/acknowledge (optional)
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 was received correctly. By sending the hash of the message, the original sender can compare it against the hash of the original message, and affirm it was sent correctly. This should be done when the message is actually seen.

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

/getPublicKey (optional)
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**)
Returns: error (numerical error code (see below), **required**)
pubkey (hexstring of public key, **required**)

/handshake (optional)
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**)
encryption (integer (see below), **required**)
Returns: error (numerical error code (see below), **required**)
message (decrypted string, **required**)

/getProfile (compulsory)

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: sender (username string, **required**)

Returns: username (username string, **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 extended character support)
encryption (integer (see below), optional, for data security)

/receiveFile (compulsory)

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, optional)
encryption (integer (see below), optional, for data security)
hash (hexstring (see below), optional, for integrity checking)

Returns: <numerical error code>

/retrieveMessages (optional)

This API allows a requesting client that has recently logged on to ask this client for any messages that were intended for the requesting client when they were offline. This client should then use the requesting client's /receiveMessage and /receiveFile APIs to pass on any necessary messages. This allows re-use of the existing push APIs, but allows the requesting client to control when that pushing occurs.

Parameters: requestor (username string, **required**)

Returns: <numerical error code>

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

Parameters: username (username string, **required**)
encryption (integer (see below), optional)
json (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)
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. Clients should report at least once per minute, but not much more than that – rate limiting may be appropriate.

Modified from https://en.wikipedia.org/wiki/RSA_%28cryptosystem%29#Signing_messages:
Suppose Alice wants to verify her identity to Bob. She can use her own private key to do so. She makes an arbitrary passphrase, and then produces a hash value of that passphrase (using SHA-512), encrypts the hash using her private key (using RSA-1024), and sends the passphrase and signature to Bob. When Bob receives the passphrase and signature, he uses the same hash algorithm in conjunction with Alice's public key. He compares the resulting hash value with the message's actual hash value. If the two agree, he knows that the author of the passphrase was in possession of Alice's private key, and that the passphrase has not been tampered with since. No certificates are used, which is less secure, but sufficient for our prototyping purposes.

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>

Group Messaging:
Once a GroupID has been assigned, all messages for that group simply include the GroupID as one of the parameters, as well as a list of recipients. Anyone can create a new GroupID at any time, by generating a SHA-256 has of the concatenated usernames (for example, if the users are “abcd123”, “abcd234”, and “abcd345”, we would hash the string “abcd123abcd234abcd345”). This GroupID should then be encrypted with each individual’s public key before sending it out to each user. The parentGroupID is the GroupID of a single preceding group that existed before this one, and presumably we want the user to be able to see the messages from the parent group.

/receiveGroupMessage (optional)

Parameters: sender (username string, **required**)
groupID (hashed ID string, **required**)
parentGroupID (hashed ID string, optional)
destinations (comma separated usernames (with no spaces), string, **required**)
message (string, **required**)
markdown (1 if the message is in markdown format, otherwise 0)
stamp (seconds since epoch time float, optional)
encoding (integer (see below), optional, for extended character support)
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)

Returns: <numerical error code>

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 Standard Not Supported
9: Encryption Standard Not Supported
10: Hashing Standard Not Supported

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
PassLib 1.7 (untested)
pytest 3.0 (and its dependencies)
Markdown 2.6

Encoding:
By default, ASCII is assumed. For any set of data, there is an optional **encoding** argument. If there is an **encoding** argument, the following encoding standards are used:

0 – ASCII
1 – ANSI (Windows-1252)
2 – UTF-8
3 – UTF-16

If there is an **encoding** argument (and the value is larger than 0), it is assumed that **all relevant arguments** in the API are call are encoded with that standard. If a client receives a message that is encoded by a standard that it does not support, it should return the relevant error message (Err 8).

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 = "01101001")
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)

If there is an **encryption** argument (and the value is larger than 0), it is assumed that **all other arguments, except for destinations,** 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).

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 binary if necessary))
3 – SHA-512 (with no salt)
4 – SHA-512 (message/file concatenated with sender username (in binary if necessary))
5 – bcrypt (with no salt)
6 – bcrypt (message/file concatenated with sender username (in binary if necessary))
7 – scrypt (with no salt)
8 – scrypt (message/file concatenated with sender username (in binary if necessary))

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).