COMPSYS302 – Login Server APIs

This is not a final document – more APIs and/or details may be added as additional functionality is added. The login server has a list of user credentials and maintains a list of currently online clients. It does not store or route any messages or any other data. **All communication with the server is over straight HTTP – no JSON encoding/decoding required (unless otherwise specified)**.
These APIs are accessible using **cs302.pythonanywhere.com/<API>?<ARG>=<VAL>**

/listAPI [GET]
Lists the APIs supported by the server as defined in this document.
Parameters: None
Returns: A list of APIs and parameters

/listUsers [GET]
Lists all of the users currently registered in the login server.
Paramters: None
Returns: A comma separated list of usernames

/report [POST]
Clients should use this API to "login" to the server and report their location. The client should report at least once every minute. If a client does not report within five minutes, they are automatically logged off/removed from the list. The client must provide valid credentials to report. Clients should report before requesting the client list via /getList.
Parameters: username
password (hexstring of hashed password)
location (integer, see below)
ip
port
pubkey (optional RSA-1024 public key)
enc (1 if using encryption described below)
Returns: <Error Code>, <Human-Readable Error Message>

/logoff [POST]
Clients should use this API to tell the server 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
password (hexstring of hashed password)
enc (1 if using encryption described below)
Returns: <Error Code>, <Human-Readable Error Message>

/getList [GET]
Allows the client to retrieve a list of all known clients in the network. The client must provide valid credentials to receive the list.
Parameters: username
password (hexstring of hashed password)
enc (1 if using encryption described below)
json (1 if you prefer a JSON list instead of plain text, optional)
Returns: <Error Code>, <Human-Readable Error Message>
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 the Error Code is 0 and json=1, a JSON dictionary is returned instead with the keys: username, location, ip, port, lastLogin, publicKey

/setGroup [POST]
Allows a client to create a "group chat" by providing a list of usernames to connect to (including the client creating the group). The server responds with a 16-character hexadecimal groupID that should be provided to all users in the group (by sending messages to them with a groupID argument) and allows the other members of the group to query the login server for the membership of the group. The server does not check if the provided usernames are valid or not.

Parameters: username
password (hexstring of hashed password)
groupMembers (comma separated list of usernames)
enc (1 if using encryption described below)

Returns: <Error Code>, <Human-Readable Error Message>
If the Error Code is 0, it is followed by a generated 12-character groupID

/getGroup [GET]
Allows a client to retrieve the list of members of a group chat.

Parameters: username
password (hexstring of hashed password)
groupID (12-character string)
enc (1 if using encryption described below)

Returns: <Error Code>, <Human-Readable Error Message>
If the Error Code is 0, it is followed by the groupID and a comma separated list of usernames

Error codes and messages:
0, <Action was successful>
1, Missing compulsory field
2, Unauthenticated user
3, False IP address or location reported
4, Invalid username
5, <Deprecated>
6, Decryption failed
7, API limit reached
8, Invalid field value
9, Invalid public key
10, Group does not exist

Credentials:
The username for each user is their University Personal Identifier (UPI).
The password for each user has been e-mailed out separately.
The password stored on the server is hashed using the SHA-256 hashing standard (using the hashlib module in Python), where the hashed text is the concatenation of password+SALT, where the salt is the username, and the hash is stored as a hexstring (a string containing the hash in hexadecimal).

To help you test if your hashing algorithm is correct, the password "thisisapassword" without a salt should hash to "a8d20c7aa37322429826e1f3cfabb08bb02d2b2bc13c12b2272cc23ff4389692".
Do not send your raw password in plaintext to the login server - you must hash it first. You should also avoid storing your raw password in plaintext within your client application.

There is no account registration in this prototype system because all users are known at initialisation time, and the administrator can statically create usernames and passwords for everyone. Similarly, for the purposes of the prototype, there is no API to change the password.

Location:
There are various issues with certain IP addresses being able to connect to other IP addresses depending on the firewall and network configurations. The **location** argument states which type of location the user is currently in. If no argument is given, the server will return an error. This is because with the issues between external and local IP addresses on the network, it is not possible for the server to automatically determine your local IP address. However, the login server will check that your reported IP address is within the expected range.

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

When reading the list of online users, it may be important to differentiate between computers in different locations (for the purposes of development only) as the location identifies which other computers will be able to connect to it. This may effectively mean that only computers in the same "location" can communicate with each other both ways.

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.

Rate Limiting:
Apart from the requirements for regular reporting, API calls are restricted for each set of credentials to 10 per minute. If the number of requests exceeds this limit, error 7 will be returned.

Encryption:
If the enc argument is 1, **all other arguments** are assumed to be encrypted.
The server uses the AES-256 encryption standard (from PyCrypto), where key="150ecd12d550d05ad83f18328e536f53", the block size is 16 (padding with spaces), and the mode is CBC. It is assumed that the iv is concatenated to the **beginning** of the ciphertext before encoding it into a hexstring, i.e. the first 16 bytes of the decoded hexstring should be the iv.

The decryption is done using this function:

```
def AESdecrypt(enc):
    enc = binascii.unhexlify(enc)
    iv = enc[:16]
    cipher = AES.new(LOGIN_SERVER_PUBLIC_KEY, AES.MODE_CBC, iv )
    return cipher.decrypt(enc[16:]).rstrip(PADDING)
```

All encrypted content should be passed to the server as a hexstring. All data is returned in plaintext. Note that special characters such as =, /, and + need to be converted to their URL encoding equivalents (http://en.wikipedia.org/wiki/Percent-encoding) before being transmitted over HTTP. By default, the server assumes that there is no encryption.