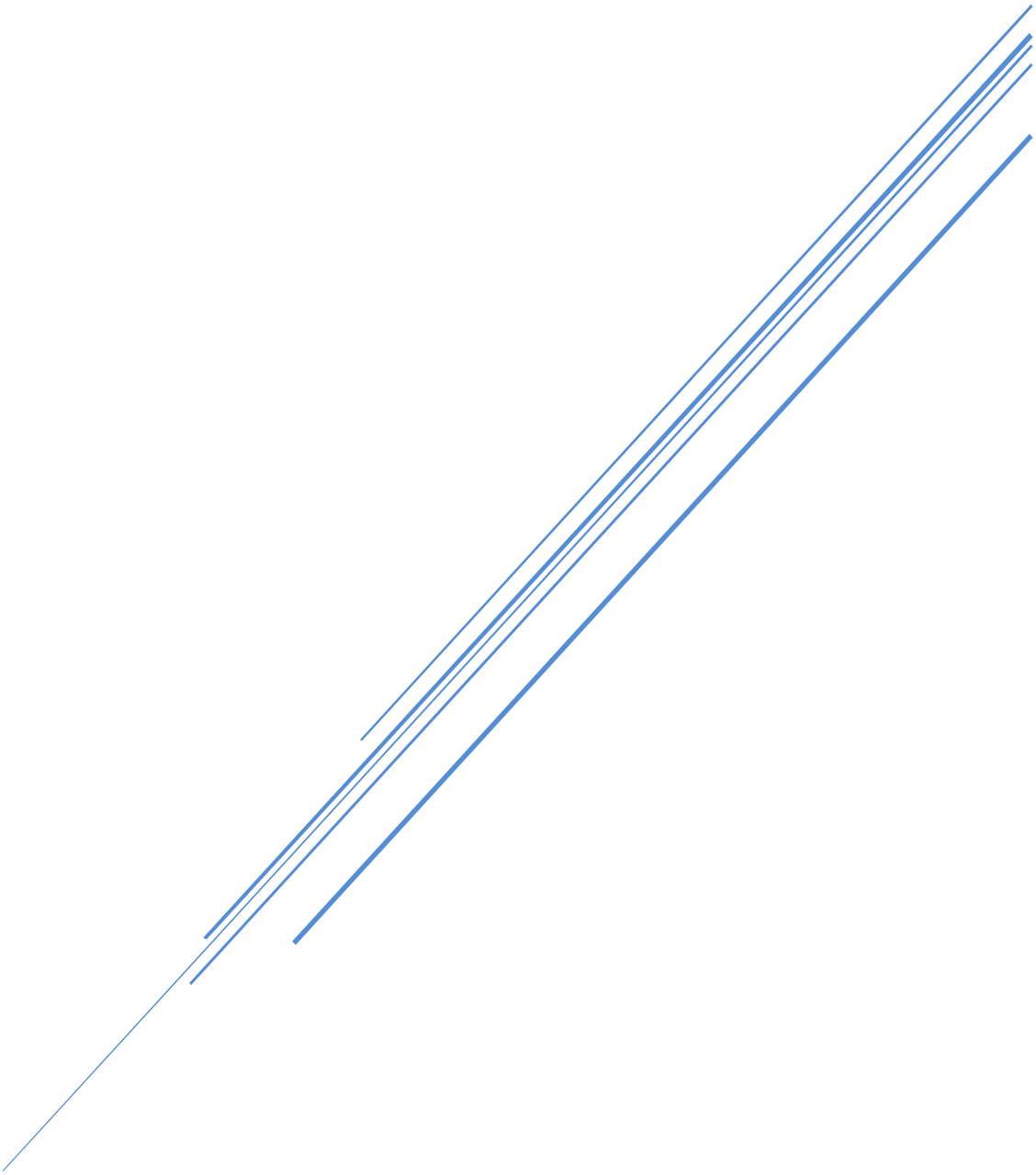
ANONYMOUS-MOOSE

Technical document for the Anonymous-moose Project



Thomas Wilson
02/03/2015

Contents

1. Intercommunication specification

This section of the document describes the packets sent between the server and clients. In each subsection one can find templates with the following information. Sent by - indicates who would send the packet and Values - indicates the values that are contained within the packet. Underneath the first line the values are listed with appropriate information.

1.1. Command types

1.1.1. ring

Sent by: server, values: group, ID
group: Ringing group - Not yet implemented.
ID: Identification number of the ring event.

1.1.2. ringACK

Sent by: client, values: UUID, ID:
UUID: Unique identifier of client
ID: Identification number of the ring event.

1.1.3. ringACKACK

Sent by: server, values: UUID, ID
UUID: Unique identifier of client that got the address association
ID: Identification number of the ring event.

1.1.4. status

Sent by: server, values: services
services: list of dictionaries containing name and status fields.

1.1.5. msg

Sent by: server and client, values: msg, I/O, UUID
msg: List of message bodys containing text.
I/O: Direction of the message: in, out
UUID: Unique identifier of associated client

1.1.6. key

Sent by: client, values: key, UUID
UUID: Unique identifier of client
key: RSA public key of client

1.1.7. disconnect

Sent by: client, values: UUID
UUID: Unique identifier of client being disconnected

1.1.8. probe

Sent by: server, values: UUID
UUID: Unique identifier of recipient client

1.1.9. probeACK
Sent by: client, values: UUID
UUID: Unique identifier of responding client

1.1.10. disassociate
Sent by: client, values: UUID
UUID: Unique identifier of client

1.1.11. chat_state
Sent by: server and client, values: UUID, I/O, state
I/O: Direction of the message: in, out
UUID: Unique identifier of associated client
state: chat state ('composing', 'paused')

1.1.12. server_online
Sent by: server

1.2. Packet structure
Packets are json encoded dictionaries including the type of the command and related values, as shown in Figure 1. With the introduction of encryption a raw packet format is used to transfer data to and from the server/client. The json encoded packet described above is split into 214 byte blocks, encrypted and collated into a list. This list along with the UUID of sender/recipient and a 'to' attribute form a dictionary. This dictionary is json encoded to form a raw packet as shown in Figure 2. The 'to' attribute is included to stop the server decrypting messages that are bounced back when a client unexpectedly goes off-line.

$$\{"type": "ringACK", "UUID": " - uuid - ", "ID": "0" \}$$

Figure 1 ringACK packet to be encrypted

$$\{"messages": ["encrypted\_block", \quad \ldots], "UUID": " - uuid - ", "to": "alice/bob"\}$$

Figure 2 Raw packet

For code examples please see the appendix.

1.2.1. Encryption
The system currently uses 2048 bit RSA keys, PKCS#1 v1.5 cipher and OAEP padding. The key length may change, but the cipher and padding are fixed. In the future 4096 bit keys may be used.

For an encryption example please see the appendix

2. Client behaviour
Apart from the 'key', 'disassociate' and 'disconnection' commands, clients only respond to incoming commands.

2.1. Start up

On start up the client sends out a key command to the server, this performs two roles:

1. Adds the client to the server's pool of connected clients,
2. Passes the clients RSA public key to the server.

2.2. Ring event

Upon receiving a 'ring' command the client asks the user to accept to ignore the ring event. If the client accepts the ring event then a 'ringACK' command is sent to the server. If the clients 'ringACK' command reaches the server first this will make the server associate the client with the user. If the client ignores the ring event then no action is taken. 'ringACKACK' commands tie into ring events. They indicate which client got associated with the user. This information is used by the client to open a chat window or tell the client operator that they did not win an association.

2.3. Status update

'status' commands are sent by the server to all clients. They contain information about the current states of the service connections on the server. The command can be sent anytime a service connection status changes on the server.

2.4. Probe presence check

'probe' commands are sent from the server if the client has not sent a command for a certain period of time (default is 7 seconds). Upon receiving a 'probe' the client should respond with a 'probeACK' within (default 3) seconds to reset the timer and prevent the client being removed from the server's pool of connected clients.

The clients also utilise probe commands to determine if the server is online. If no probe commands are sent to the client for more than X seconds then the client should notify the user and begin sending key command to the server address every X seconds. When the server comes back online it will receive one of these key commands and send a status update.

2.5. Messages

'msg' commands carry text messages to and from the client and server. Server to client have I/O set to 'in', client to server have I/O set to 'out'.

2.6. End of chat

'disassociate' commands are sent by the client to end an association between the client and an user. Example usage: client operator closes the chat window, thus sending a 'disassociate' command

2.7. Client closing

'disconnect' commands are sent by the client to remove the client from the server's pool of connected clients.

3. Continuing support

At some point Thomas Wilson will leave Lancaster and someone else will need to pick up the reigns. Sections 1 and 2 of this report are primarily included to encourage others to produce clients compatible with the system.

3.1 Foreseeable development

1. Standardise the messages passed from service threads to dispatch to a similar format as the intercommunication specification. This will aid easier integration of new services in the future.
2. Make server more modular in respect to service threads. So to allow easier integration of new services in the future.
3. Add thread watchdog to server. It should ping each thread every X seconds and determine whether they're still alive. If any threads have crashed they must either be restarted or the program must restart.

Appendix 1

The graph below shows the message flows between the NL chat clients and the end users.