Network Service Design of an Instant Messaging System

This document details the network design and implementation of an Instant Messaging System (IMS). We have borrowed some ideas from the design of peer-to-peer networks, since they are well suited for this application. But the architecture is not peer-to-peer by itself; it is essentially **hybrid**. The service was designed to model an instant messaging system like **MSN**. We shall first describe the overall network architecture of the system and then delve into its implementation.

Network Architecture

The IMS proposed herein has 3 main components:

1. Central Repository Server (CRS): The CRS is, as the name suggests, like a central repository for the messaging service. It contains primarily-
    a. List of active (enrolled) servers of the IMS
    b. List of all registered users, and information pertaining to them, like password, contact list, his private-public key pair and which *client* he is logged in at (network address of the client).

    It is assumed that users have registered with the service previously (through the internet). The service has only one CRS.
2. Messaging Server (MS): The service could have several MSs, and their geographic location is not restricted either. Each MS is enrolled in the service by registering itself with the CRS. The role of an MS is to primarily authenticate clients. Thereafter the MS provides some services to the client but is not required to contribute in communication between clients. The load of users currently logged in to the service is distributed between all the MSs currently enrolled with the CRS. The design does not expect an MS to be a high-end host; a more powerful host could perhaps support more concurrent clients, but this is not a requirement imposed by the architecture.

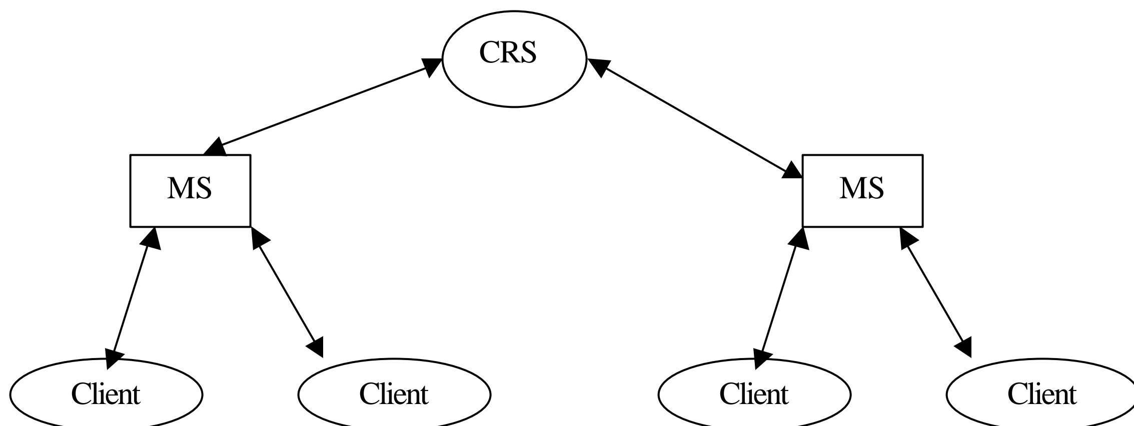

Fig. 1: Network Architecture

3. Client: The client is the end-user software that communicates with an MS. The software is to be downloaded and run on the user's host machine. It is expected to have an internet connection, and the ability to exchange text (ASCII) based TCP packets.

Terminology

In addition to the three terms defined above, we introduce the following terms –

- **User**: A user logs into the IMS via the client. The user is expected to register himself with the service prior to signing in. Once registered, the user has a unique username by which to identify himself, and a password to authenticate login.
- **Contact-list (a.k.a. buddy-list)**: Each user has a contact-list which is a list of other users whom he considers as online friends (a.k.a. buddies). Once Joe has signed into the service, every time one of his buddies signs in or out, Joe's client (and hence Joe too) is kept updated of the change.
- **Message**: A data packet sent from an online user to his buddy (who is also online). The service does not support sending offline messages i.e. a message from an online user to his buddy who is offline. The content of the message is not restricted by the architecture i.e. it could be text, file transfer or MIME.

Other key features of this service are-

- A user can send a message to only an online user in his contact-list.
- Currently, the service does not support the creation of groups/channels. Users can send messages to multiple users, but there is no group creation, group theme etc. One-to-many messages are not supported either, but extending the current design to handle it is not difficult.
- At a time, only one client can be run on a host.

Design Issues

The following design issues were deemed important for this IMS-

- **Scalability**: The service was designed keeping scalability in mind, and is expected to be able to support as many users as required, so long as sufficient MSs are maintained to service them. The CRS itself does not place a bound on the number of MSs active concurrently, or the number of users that each MS can service. The CRS is expected to provide information about all registered users, and this could be a large number. However, this should not be a constraint on the number of possible users, since the CRS itself could be a distributed database. The author felt that this particular aspect was not critical to the design of the system, and is hence not elaborated upon. More details follow in the section "How it works".
- **Security**: Security was a prime concern in the design of the system, and the service is expected to provide secure and robust communication. The service uses *secret key* as well as *public key cryptography* to provide the same. Secret key cryptography is simpler and faster than public key cryptography, and serves the purpose well in most

cases. Public key cryptography is used only to exchange the secret session key between clients, and thereafter secret key communication follows. All pertinent communication is encrypted, lending this service well to use in business and corporate establishments.

- **Reliability**: The service provides means for reliable communication between users. The underlying transport protocol used is TCP (Transmission Control Protocol) and the service utilizes the in-built reliability of TCP.

How it works

This section provides insight into how the service works, and is hence the most critical one of the document. We start by defining the precise role of each component of the system.

Central Repository Server

The Central Repository Server (CRS) is the core of the service. It holds details of all registered users, and messaging servers (MS) currently enrolled with the service. The CRS interacts with MSs as required, providing them with the required information.

- The CRS shares a secret key (MSkey) with all MSs. Since an MS would also have to be started up by personnel who are part of the IMS, this sharing is not expected to be a problem.

As described earlier, the CRS stores along with each username:

- his password, contact-list, profile information etc
- if the user is logged in, his client's network address
- each user is allotted a public key-private key pair. These keys are used for providing secure communication, and are stored with the CRS.

The CRS is also responsible for providing clients with hostnames of active MS in the service.

Messaging Server

A Messaging Server (MS) is a host which services the clients' requests. An MS need not be confined to any geographic location, all that is required is for each MS to be able to connect to the CRS, and listen for clients' requests. These client requests may be of several kinds: a login request, logout request, change profile/password request etc. It is reiterated that an MS does not participate in the actual transfer of messages from one user to another. Messages are sent between user-clients directly, and such a design provides for good scalability (since the MS is not involved in communication that it actually need not be involved in).

As part of its startup configuration settings, an MS reads in (from a file) the network address (IP address and port) of the CRS and the MSkey. The host then attempts to contact the CRS, to enroll itself as an MS. The request comprises:

- network address of the MS (sent as part of TCP anyway)
- MSkey sent encrypted by itself i.e. MSkey is used to encrypt itself and the result is sent along with the request
- The CRS, upon receiving a request, decrypts the request using its copy of the MSkey, and if the decrypted value matches its stored copy, authentication is successful and the host is enrolled as an MS into the service.
- Each MS stores the list of users currently logged in with it, and each user's contact-list. The MS needs to know each user's contact-list so it can pass on any changes in the status of any of the user's buddies.

The above procedure provides a robust and secure means for hosts to enroll themselves into the service. At this point, the MS just waits for user requests at a pre assigned port. All communication between MS and CRS takes place through single key encryption using the MSkey.

Once an MS is enrolled into the service, it essentially becomes a trusted host. The design follows the policy of trusting hosts vs. a Kerberos-based approach (wherein each request is authenticated) to keep the network traffic and response time within feasible bounds. Authenticating each request from a host could considerably slow down the response time. Also, trusting an enrolled MS would suffice if the MS were itself not left open to a security breach.

Client

The user must download and run client software (hereafter referred to as "client") which shall interface with the rest of the IMS. The client, upon startup, reads in the host-name of the CRS. The client goes on to contact the CRS to obtain a list of MSs to contact. The CRS would resolve these requests so as to balance the load evenly among all enrolled MSs. If each MS has a different configuration, then the load balancing could be determined so as to not exceed any server's maximum capacity. Communication between a client and its MS is encrypted using the user's password (secret key encryption). This shall do because only the client and servers of the service (CRS and the other MSs) are aware of his password.

Login process:

- Once the client determines the address of the server (MS) it should contact, it goes on to contact the MS (let's call it user-MS for convenience), the transport protocol of this exchange and all subsequent ones again being TCP.
- The client sends the username of the user (say Joe) in plaintext, and the password encrypted by his password itself (akin to secret key encryption). Once the user-MS gets the login request, it proceeds to contact CRS to enquire if Joe is logged in at any of those servers. To achieve this, the user-MS (attempts to) obtain a mutex lock on Joe's record: if it cannot (perhaps because another MS has already locked the record) the login request is denied; if the user is already logged in, the login request is again denied; else the login request is permitted to go through to the next step (the mutex lock is not released yet). This exchange between user-MS and CRS is ecrypted using the MSkey.

- The method outlined above seeks to prevent a user from logging in at two terminals. If two nearly-concurrent or concurrent requests arrive from two different user-MS for the same user (Joe), the first user-MS shall get the mutex lock while all successive ones won't, thereby resolving concurrent login requests. If login is successful, the user-MS goes on to the next step.
- The user-MS contacts the CRS to look up the password for that username, decrypts the given password, and compares the obtained (decrypted) value to the password provided by CRS (this exchange between CRS and MS shall again be encrypted using the MSkey). If the passwords match, the login process proceeds to the next step, else a "password failure" message is sent back to the user.
- After login succeeds, the mutex lock is now released. Joe's client is informed of the login success. The next crucial step is to give a copy of Joe's private key to his client. Joe's user-MS gets a copy of his private key, encrypts it with Joe's password and the result is sent to Joe's client. Joe's client upon receipt of the encrypted private key, stores it temporarily in memory.
- Joe's user-MS then looks up the CRS for his contact-list (including each online buddy's public key), stores a copy for itself and passes on this list to Joe's client. Any pending requests by other users to add Joe as a buddy (see "User Actions" below) are also relayed to Joe's client for approval.
- The user-MS updates the CRS to now hold Joe's client-address. This update in Joe's record at the CRS is propagated ("*pushed*") to all MSs by the CRS. Upon receiving such an update from the CRS, each MS checks to see if any user logged in with it has Joe in his contact-list- if yes then the update is pushed to that client as well. This "update" includes Joe's username and his public key (both are sent in plaintext).

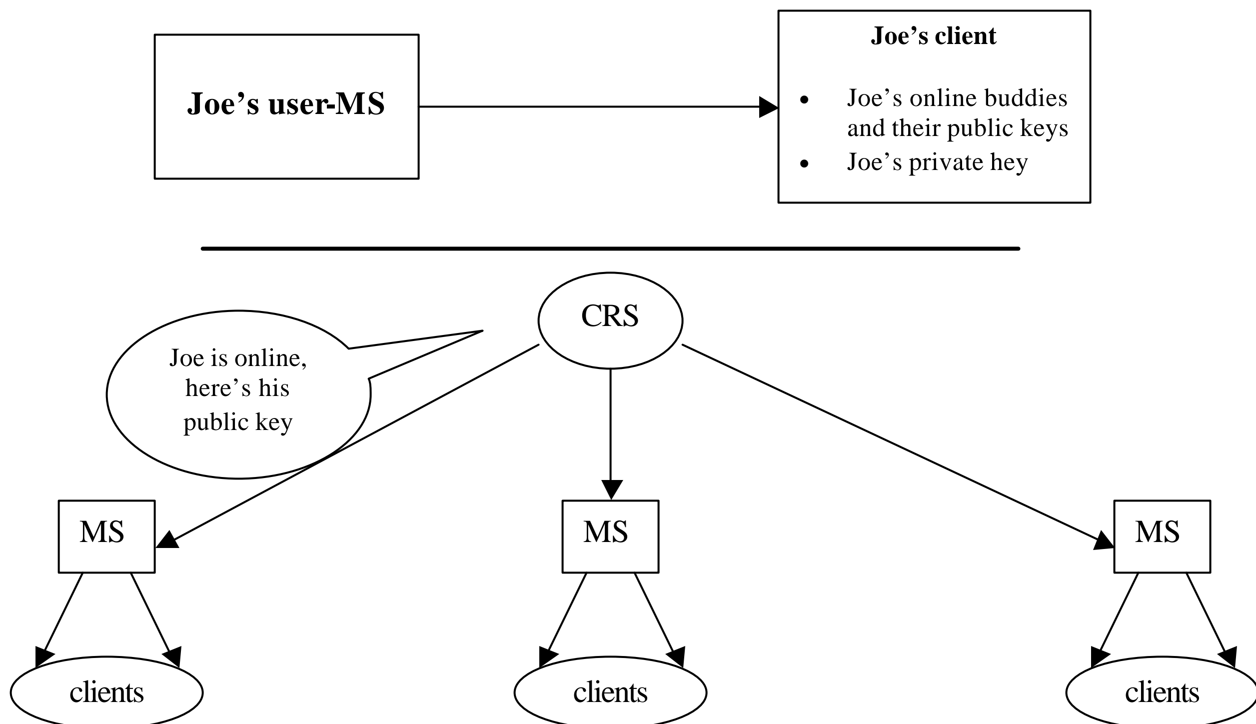

Figure 2: Joe is now online

Upon completion of this login process, Joe has been identified as an online user in the service, and Joe's client knows the network address of each of his online buddies. Also, every online user having Joe in his contact-list also knows the address of Joe's client. The above figure seeks to explain the what transpires during the login process.

Communication between Clients

- At the end of the login process, Joe is aware of all of his online buddies (and their public keys) and all of his buddies know his client's address and his public key. Also, Joe is given his private key.
- At this point, any client can initiate communication with any other buddy's client. Say Joe wants to send a message to Alice, who is also online. Joe's client sends a INIT request to Alice's client, encrypted by Alice's public key. The request comprises a secret *session-key*, which shall be used for communication between Alice and Joe for this session. TCP ensures that this shall be delivered to Alice (or an error reported), and the encryption ensures that no eavesdropper can make sense of the request. Alice's client receives the request, and after decryption gathers that it's an INIT request from Joe. Alice acknowledges the receipt of the session key by sending a INIT-ACK to Joe using his public key. All future communication between Alice and Joe is now encrypted using the session-key. This session key is to be generated randomly by Joe/Alice's client.
- The session-key is invalidated when one of the parties involved logs out.

As described above, the service utilizes encryption to ensure secure transfer of messages.

User Actions

The possible actions an online user can take are:

- Communicate with another online buddy (as detailed above). This communication is not restricted to text messages; it could even be files. The protocol of exchanging a secret session-key remains same.
- Add another user as a buddy. To add Alice to his buddy-list, Joe needs to be able to search for her in the database maintained by the CRS (based on username, profile information etc). Once Joe has identified Alice's username, if Alice is online, an ADD request is sent from Joe to Alice in plaintext. Alice's client receives the message and can at this point either grant or deny the request. If granted, Joe's client updates his user-MS about the change, which goes on to retrieve Alice's public key from the CRS and send it to Joe. If Alice is not online at the time Joe sends the ADD request, the request is stored in the CRS along with Alice's profile information and is sent to her client the next time she logs in.
- Remove a user from buddy-list. This is rather straight forward- to remove Alice from his list, Joe sends a REM request on Alice to his user-MS, which updates the CRS.

Implementation details

The transport protocol used throughout the service is TCP, in keeping with the reliability requirements. TCP is *connection-oriented* i.e. one socket can only be used to communicate with one address. The protocol (so to say) that shall be followed here is:

- The client listens on its well-known port (affixed by the IMS) for new data. This data could come from its user-MS (update on a buddy logging in/out) or a request from another client to exchange messages.
- The client shall be pre-threaded: the reason for choosing threads is that they are "lighter" than forking; and it would be pre-threaded to avoid threading on demand which causes some slowdown in performance. Each new request from a user-MS or a client shall be handled by a new thread.
- All servers of the IMS (CRS and MS) are also pre-threaded.

Application Layer Protocol

The application layer protocol used for all communication of the service is akin to HTTP-style requests/responses. All data is sent in ASCII text format (8-byte characters). To transfer audio/video/other files, the service utilizes the MIME format. Each line (upto and including the blank line after headers) ends with a CRLF ("\r\n"). A message looks like:

method
headers

-- data (if any) --

For instance, a login request looks like:

LOGIN
Username: Joe
Password: pass *(in encrypted format)*

A request from Joe to Alice to start a conversation looks like:

INIT
The following is encrypted with Alice's public key
From: Joe
To: Alice
Session-key: rabbit hole

It should be evident that the request closely resembles HTTP requests. A message from Joe to Alice looks like:

MSG
The following is encrypted with the session key
From: Joe
To: Alice
Content-type: text
Content-length: 100

--actual message data--

This document does not detail all possible method-names. However, it is hoped that the above examples give a general idea of what the application protocol looks like.