The Engage Communications System:
An Independent Study in Computer Systems

Justin Weisz and Michael Piatek

July 9, 2002

1 A Brief History of Human Communications

According to the Bible, when G-d created Man, Man and G-d were able to communicate using sound. Man was able to create sounds using his biological gifts, and G-d was able to perceive and understand these sounds. Further, G-d was able to produce audible sounds which Adam was able to recognize and understand, using his ears. Then, G-d created Eve, and appointed her the task of naming animals. This, or perhaps some other explanation involving Darwinism and evolution, form the beginnings of communication among human beings.

Over time, Man began to seek and use other forms of communication, for various other purposes. Examples include, and are not limited to: facial expressions, bodily gestures, pictures, and ultimately, the written word. Then, Man started figuring out methods for more quickly transmitting the written word to other Men around the world. For example, books were published, newspapers were printed, and ultimately, the telegraph was invented.

Now, Man could communicate with other Men over arbitrary distances, and Man was fascinated. But, this was not good enough, as Man wanted some means of vocally communicating with other Men across the world, in real time. So, Alexander Graham Bell invented the telephone, and Man could now transmit his voice to other Men across the world, in real time.

But now Man had another dilemma. Man had created an invention beyond his wildest dreams—a computer—which could perform significant computational tasks involving numbers in a fraction of the time it would take a Man. And, like when G-d created Man, Man wished to communicate to this computer, and have the computer communicate with Man. The keyboard and monitor were thus invented to serve these two purposes. However, Man was demanding, and wanted this newfangled computer to talk to other computers. So Man (actually a collection of Men under the guise of DARPA) created the ARPAnet, which allowed computers—and ultimately Men—to communicate with each other, in real time.

As time went on, the ARPAnet evolved into the Internet, and the Internet exploded in popularity to the point where (almost) any Man in the world can connect to it, and communicate with other Men around the world, in real time.

In the beginning, conversations among Men took place using a system called ASCII, by which characters are drawn to the screen as they are typed in. However, this was very reminiscent of when Man used the telegraph, and as with the telegraph, Man was not satisfied with merely sending characters around. Man wanted to express himself using pictures as well as text, and sound as well! So, Man invented two-dimensional graphical chat (as well as a variety of other Internet applications and protocols, but we'll stick with this one for now).

However, there were many problems with the system Man created for graphical chat. The system by which Men represented themselves with pictures was very limited. Communication among Men could only take place in a very localized space, which kept Man from talking to Other Men when they were not in that same space. Man also wanted to do more with the system, and transmit not just words, but "files". However, Man could not do that with the system he had created. So, two Men, named Justin Weisz and Michael Piatek, took it upon themselves to fix these problems, and create a system which all Men could use to communicate with all Other Men, and perhaps Women too!

2 The Vision

This section will be an attempt at providing you, the reader, with a glimpse of the vision that Mike and Justin share for the future of Internet communications as a whole, as well as the specific problems and goals we wish to solve with this project.

First and foremost, it is our fundamental belief that graphical communication provides many benefits not realized by plain text-only communications. Most Internet communications today are based on the transfer of text; this is unavoidable, as the written word is how we, as humans, communicate. However, the majority of person-to-person communications over the Internet today only use text to convey a message. This includes email, IRC, and instant messaging. There are some notable exceptions to this, as people are starting to become aware of the use of graphical symbols to express ones self on the Internet: AOL's buddy icons, graphical smileys, and HTML email with inline images.

However, what we envision is in regards to real-time communications, i.e. that of a conversation that takes place among an arbitrary number of participants. In our model, we have a notion of a shared environment, one in which people interact in, and with. There are any number of spaces, or "rooms" in the environment to which a user can travel, and meet other users. This is akin to channels on IRC, or frequencies on a HAM radio.

Associated with each of these rooms is some graphic, or animation, which provides a sense of locality for the user. This graphic represents the space or location the user is interacting in, and it can be anything a user wants. For example, if the user wants to pretend that they are at the circus, then it can be a picture of a circus ring. Or, if the user wants to travel to Paris, then it can be a picture of the Eiffel tower. However, this graphic should not be constrained to being just a static image. The user should be able to specify **any type of media** they want, where media is defined as a stream of bits which, when interpreted, produces a graphical or auditory effect, or both.

This is a very loose definition of "media", and the reason for this is because our concept of "media" keeps changing. For example, MP3s are clearly a good example of "media" today, but the MP3 standard wasn't created until the late 1980s. So, our notion of "media" must be adaptive enough to handle future types of media which have not yet been invented, but which can be easily incorporated into our implementation of this system.

The second major concept we have is that of an object. Objects should behave much like they do in the real world; that is, we should be able to experience them, and manipulate them, and they should be able to react to us, and perhaps act on their own, depending on what kind of object this is. Objects also have an associated piece of media, since they too need to be represented visually in our environment.

Finally, the third major concept we have is that of a user. Up until now, we've used the term 'user' loosely, but now we shall define it more precisely. A user is any object in the world capable of communication in this environment. Note that the words "human being" do not occur anywhere in that definition. That's partly because humans don't matter in this equation, since we also want to leave room open for autonomous scripts or bots which communicate on behalf of a human being. We could define a user as being an instance of our binary program running on a computer, but since we have not gotten to implementation yet, we're not going to define a user this way.

So, our vision is of a combination of users, rooms, and objects, all of which have associated media, and all of which can act in some way toward others. Users can manipulate objects, or even interact with the room, if the media associated with the room supports interaction. The room provides the atmosphere and context for the communication which will be taking place inside of it, and serves to partition communications amongst all users in the entire environment. Objects should be able to react and respond to users' demands, and perhaps act out of their own accord based on some prior specification.

That is our vision.

3 Independent Study Goals

We had many goals and questions we wanted to answer for this independent study. For the purpose of completeness, we will list them again here, and explain how we either met that goal, or what challenges we faced in trying to meet that goal.

- **What is the best way to manage inter-server connections to provide guaranteed, reliable, and fast message passing between clients?**

  At a basic level, message passing between servers can be achieved by viewing the servers as a graph structure, ensuring strong connectivity, and using message flooding to distribute messages. However, by using hash key based lookup systems such as Chord[1] and viewing the resource to be located as the status of a user, we can achieve message passing and user

[1] Chord's home page is at `http://www.pdos.lcs.mit.edu/chord/`

status lookups more efficiently. We can use such a system because each user is given a user identification key to be used with SSL.

- **Is it possible to guarantee message passing between clients?**

  Strong connectivity guarantees messages will fully propagate. Because Chord distributes the search space among multiple hosts, its fault tolerance is not as complete as strong connectivity, but since there is some overlap in the search space at each node, and we do not expect servers to drop very often, its error is sufficient for our purposes.

- **How do we determine client uniqueness across the network (i.e., a GUID). Will a centralized server which assigns GUIDs be necessary (i.e., ICQ), which would also require users to register their clients, or can this be done on the Engage network dynamically and transparently?**

  An EngageServer can be configured to act as a Certificate Authority, and sign certificates for users. Therefore, each client has a GUID in the form of an X.509 certificate. This system does require a centralized server (or multiple similarly configured servers), and it does require users to register their clients.

- **Create an object metaphor for objects in the graphical environment—each object has an associated media, an owner, and a set of abilities (permissions), such as can be moved, removed, downloaded, etc.**

  Objects currently have these properties. The Java Engage client does not currently implement media, other than for a default user avatar. However, as explained in section 2, we do have a very good idea of how media objects should work in our environment; it's just a matter of writing more code.

- **Create a workable graphical interface to facilitate interaction in this environment, as well as manage media within this environment.**

  The Java Engage Client has a working user interface as shown in figure 1.

- **How do we distribute large (uncached) media files between users?**

  Media file transfers are orchestrated through data requests to the server, which in turn makes data requests to the original clients possessing the media. Most media is then cached on servers where it could ideally be segmented over the server space using principles from Chord, just as user status is segmented, resulting in efficient lookup and retrieval.

- **Can we compress data before it is sent to minimize bandwidth usage (especially for large files), while also not slowing down the host computer?**

  We solved this problem in a roundabout way. OpenSSL will compress the encrypted data before it is transmitted, so technically, we achieved compression in our protocol. However, compression in general will always add some computational overhead, which will increase with the size of the data.

- **Should we encrypt every transmission and create a completely secure network? Would a public/private cryptosystem be appropriate for this kind of network?**

  Encryption is a nice side benefit to the requirement that every client has its own unique identifier. In order to uniquely identify clients, we use a public/private (asymmetric) cryptosystem, and verify that a client possesses the private key which corresponds to the certificate they

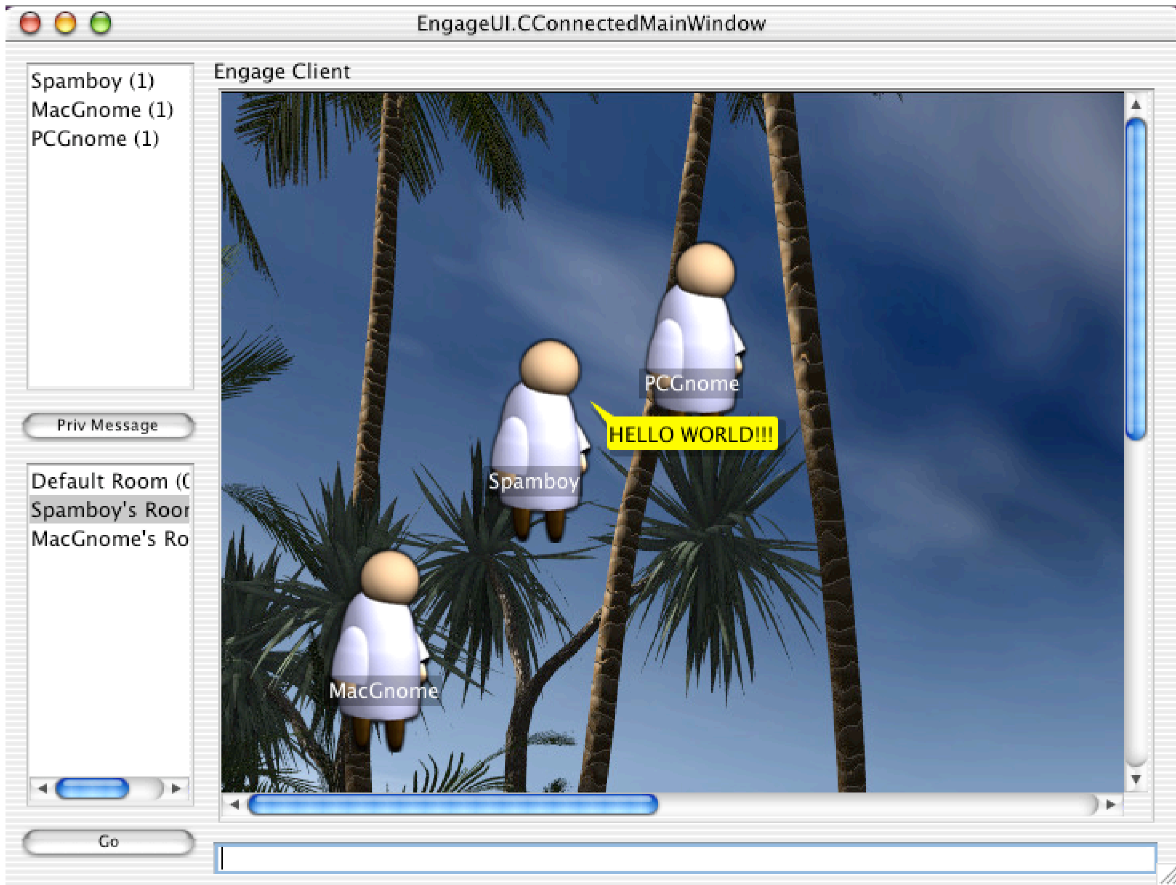


Figure 1: The Java Engage Client running on Mac OS X

present to the server. More on this is explained in sections 5.4 and 7. So, the answer to both of these questions is "yes".

- **Can [our protocol] support encryption and compression? For example, have a "compressed" flag in the packet header, and send the packet header uncompressed, but compresses the packet, and then send that. Same with an encrypted flag—should we encrypt both the header and packet, or just the packet?**

  This is not necessary, since OpenSSL already encrypts and compresses data, and we do not want to add any more computational overhead to our protocol.

- **Do we need any sort of checksumming? TCP is a guaranteed delivery protocol, but then, why do other protocols (such as ICQ) use checksumming for their TCP communications?**

  Not at all. We have checksums in the following layers: data link, TCP, SSL. We don't need any more than this, and if corrupt data manages to make its way through all three of these layers, then it probably deserves to get through! Why ICQ uses checksums was never answered, but since ICQ was converted to use the Oscar protocol, this is a moot point.

- **Create an implementation of the Engage Client in Java**

Mike did a wonderful job of this, and his implementation has been tested on Mac OS X 10.1, Andrew Linux, Red Hat Linux 7, Windows 98/2000/XP, and Solaris 5.8. The implementation requires JDK version 1.3.

- **Port the Engage Server to Linux**

  The server needed some minor modifications to compile on Red Hat Linux 7.1. We did find some issues when converting from a big endian system to a little endian system, but these were more easily handled in the Java client, so we handled them there.

4 The Engage Protocol

4.1 Overview

The Engage protocol is very simple, and runs on top of TCP. This means that packets are guaranteed to arrive, and in order. This is a necessary property, because people don't like to have their packets dropped in the midst of a conversation. Since a small amount of lag is acceptable in this application, any overhead caused by TCP does not interfere with a normal conversation.

Every network message is prefixed with a header, which is a 3-tuple of ⟨ message type, timestamp, payload size ⟩. The message type field is 2 bytes (short), and the timestamp and payload size fields are 4 bytes each (long). The message type field is used to determine what kind of message we are receiving. The timestamp field is currently unused, but can be used in the future for synchronizing clocks, if necessary. Finally, the payload size tells the receiver how much data to read after the packet header. This data is the payload of the message, and is used to perform the action as described by the message type.

It should be noted that there is a strong distinction between a request and a message. A request is a message type which demands a response from the other party. A message is a message type which does not require a response. There are no message types which have an optional response; all message types will either generate a response or not.

4.2 Message Types

Message types include the following:

- Authentication messages
- Server uptime request/message
- Message of the day request/message
- Requests/messages pertaining to client information
- Requests/messages pertaining to rooms, the state of rooms, and the media inside of rooms

- Requests/messages for media objects and data
- Chat messages, private chat messages, and broadcast (server-wide) messages
- Messages pertaining to server administration
- Messages for events such as client connect/disconnect,
- Messages pertaining to kicking off users or banning users
- Ping/pong messages

5 The Engage Server

The Engage Server is a POSIX-compliant, multi-threaded server which has been compiled and run on MacOS X 10.1, and Red Hat Linux 7.1. It is written in C and C++, and uses structures from the Standard Template Library (STL).

5.1 Requirements

EngageServer requires libpthread and openssl 0.9.2b (or newer).

5.2 Overview

EngageServer is the heart of the Engage Communications System. It is responsible for hosting a single virtual environment, segmented into multiple rooms, each with it's own look and feel as defined by the user. The EngageServer binary runs on a single machine, and clients connect to a single instance of this program. Each client is processed in its own thread. The EngageServer is then responsible for the forwarding of all relevant network activity to the client, based on what occurs in the environment. For example, if one user speaks a chat message, that client will send it's message to the server, and the server will multicast that message over to all clients who are within the scope of that message (i.e., in the same room).

The EngageServer is also responsible for creating and maintaining active network connections to other EngageServers, thereby forming The Engage Network. An overview of this network is provided in section 7.

5.3 Features

- Client authentication with SSL certificates

- Support for user accounts with different permissions (e.g. ability to talk, wear media, move media, admin, etc.)
- Support for uptime requests, and a message of the day
- Room support, with a URL for background media
- Hotspot support, allowing the user to store generic data in a rectangular shape inside of a room (i.e. scripts or doors)
- Support for user worn media
- Media caching
- Chat in rooms, private messages, and broadcast messages (to everyone on the server)
- Support for remote administration of the server
- Support for connecting and communicating with to other servers with SSL authentication
- Support for ping/pong messages to detect clients who have timed out
- Support for banning users for a time interval based on IP or user account

5.4 Security

In both the Engage server and client, we use SSL for all communications. This provides us with a robust, secure layer in between the Engage protocol and TCP which encrypts and compresses our data before it is sent over the wire. There are two major reasons for why we needed this functionality, and those are listed in the next section.

5.4.1 Why OpenSSL?

OpenSSL is an open source implementation of the Secure Sockets Layer version 2 (SSLv3), version 3 (SSLv3), and Transport Layer Security version 1 (TLSv1) protocols. These protocols provide an extra layer between the application and TCP, and provide encryption and compression of application data before it is sent through TCP. OpenSSL also provides support for the X.509 certificate standard, which allows certificate signing.

There are two main reasons we use OpenSSL in our system. The first is that it provides each client with a unique public certificate and private key, and the second is that it provides transparent encryption and compression in our transmissions. However, these benefits do some with a small price, as there is some computational overhead in performing encryption and compression. We have not measured this overhead explicitly, but the overhead is a necessary evil since we absolutely need each client to have its own unique identifier. More on this is in sections 5.4.2 and 7.2.

5.4.2 Certificates

Every instance of the Engage Client and Engage Server has its own public certificate (in the X.509 format) and corresponding private key. There is also a master Engage Network private key, with corresponding self-signed certificate, which is used to run a certificate authority. In order for a server to join the Engage Network, or for a client to connect to a server, their certificate must be signed by this authority. Certificate signing is an automated process in the Engage Server, and is triggered by a connection to a well known port (currently 3665). This process is explained in detail in section 5.4.4.

When the Engage Client connects to an Engage Server, the client sends its signed certificate as part of the SSL handshake. The server then sees that they both share a common trusted certificate, the Engage Network Root Certificate, and then finishes the SSL handshake. Then, the server looks up in its local database of users to find a user account matching the certificate the client sent. If it finds one, it grants that user all of the permissions specified by that user account (i.e. server administration permissions); otherwise it uses the default "guest" permissions (i.e. chat, wear media, etc.).

This method guarantees that the user actually has the private key corresponding to the certificate they present to the server. Therefore the only way to hijack someones account on the server, and perhaps gain server administration control, is if the user's private key is stolen. Otherwise, a user cannot use another user's certificate to pretend that they are that user, because they do not have the corresponding private key, and will not pass the SSL handshake.

5.4.3 Certificate Authority

As mentioned before, there is a master Engage Network Root Certificate, which is used to sign all other Engage certificates. This is necessary because two parties need to explicitly trust a third party in order to establish a secure connection. In order to manage certificates, an EngageServer can be configured to run as a certificate authority if the root certificate is placed in the same directory as the binary. If a file named `engagenetwork.key` is present, an MD5 checksum is performed on this file. If the resulting checksum matches the actual checksum of the Engage Network Root Certificate, then that server configures itself to become a certificate authority, and automatically sign certificate requests when they are sent by a client. The checksums are performed to make sure that the `engagenetwork.key` file isn't replaced with some other master key, ensuring that automatic certificate signing takes place only in the presence of the real master key.

5.4.4 Automatic CSR signing

When a server is configured to become a certificate authority, it opens a socket and listens on port 3665 for incoming connections. When it receives a connection, it spawns a new thread, which will read in a Certificate Signing Request (CSR). The server then verifies that a valid CSR was received, and runs a shell script which produces a signed certificate. This shell script was adapted

from Apache's `mod_ssl`, and it provides a nice wrapper for using the `openssl` command line tool with a custom script for performing CA operations. Finally, the server will read in the certificate created by this script, send it back to the client, and close the connection.

Now, the client has a valid Engage Network certificate which can be used to connect to Engage Servers. It should be noted that this is a necessary step before the client connects to any servers, and can be automated in the client as well, thus becoming an almost transparent process to the user.

6 The Engage Client

The Engage Client is a 100% pure Java application with separate front / back end code bases that communicate via a standardized callback interface class defined by the back end. This separated back end allows the pure Java networking code to be migrated to various platforms and have native interfaces built on top of it.

Endian issues are handled via a `ByteOrder` class. As network data arrives, its header is parsed and its payload collected as a raw byte array. Message subclasses unpack this data using the aforementioned class, which can change endianess based on a flag.

The existing Java interface uses only Swing widgets and pure Java constructs for maximum portability. Swing was chosen over the Abstract Window Toolkit (AWT) because of its more strict platform independence. Also, most active Java widget design is occurring in Swing.

7 The Engage Network

The Engage Network is a network of Engage servers responsible for routing messages between servers. Currently, the network is set up as a complete graph on n nodes, where n is the number of Engage servers on the network. This network is currently used to broadcast messages pertaining to clients joining and leaving particular servers, as well as servers joining or leaving the network. Future plans for the Engage Network are detailed in section 7.4.

7.1 Protocol

The Engage Network protocol is used by Engage Servers to communicate with one another. This protocol runs on top of TCP, and uses port 3664. The protocol uses the same packet header as explained in section 4.1. Message types currently include the following:

- Server joins the network
- Server leaves the network

- Client joins the network
- Client leaves the network
- Ping and pong messages for timeout/crash detection

Servers also use SSL when communicating with each other. This was a very easy feature to implement, since each Engage Server already has a certificate signed with the Engage Network root certificate. Again, we feel that the security benefits provided by using SSL outweigh any performance losses experienced.

7.2 Why Clients Need Unique Identifiers (or, Why OpenSSL? Part II)

The main purpose of the Engage Network is to inform clients of when their friends come online on other servers. In order to do this, each client maintains a list of his own friends, and the network will send all clients messages when a client joins or leaves the network. Then, clients can tell when their friends come online by matching the network messages against their own friend list.

In order to do this, we need some way of differentiating clients on the network, since each client has to store a unique identifier for each of their friends. We considered several solutions to this, and they are listed below:

- **Use a client's nickname as a unique identifier.**

  This is exactly how a system such as ICQ or AIM works. Each client is assigned a unique name or numerical identifier, and then clients can store their friends' nicknames in a list format. However, there are three significant drawbacks to this solution. The first is that we wish to allow a client to change his nickname at whim, which means that two friends would have to add all of the nicknames they wish to use to each others list. The second is that it is very simple for a malicious user to pretend they are someone else, which is something that should not be allowed. The third is that different users may wish to use the same nickname, thereby causing a namespace conflict, and all sorts of problems for both the users and the system.

- **Use a client's IP address as a unique identifier.**

  This solution was very tempting, until we realized that not all clients have a static IP address. So, it would be impossible for clients to keep track of their friends when their friends' IP addresses kept changing every time they logged on to the system.

- **Use the MAC address from a client's Ethernet card.**

  Every Ethernet card manufactured carries with it a unique 48-bit IEEE MAC address. This seemed like an ideal situation, since MAC addresses are guaranteed to be unique. However, we then realized several significant problems with using MAC addresses. First of all, an Ethernet card would be required to use the Engage Network. However, many lower-end home PCs do not have Ethernet cards, since they use modems to connect to the Internet. Second, some Ethernet cards allow the MAC address to be changed via software (such as wireless

Ethernet cards), which means that spoofing can still occur. Third, using a MAC address would tie the Engage Network identity to a particular Ethernet card, meaning that it would not be portable at all (i.e. if you upgrade your Ethernet card then you lose your identity and have to let all of your friends know that it changed). So, while this initially seemed like a good solution, it did not meet our needs.

- **Use a public/private key cryptosystem.**

  This is the solution we chose. In this instance, a client generates their own public certificate and private key, based on randomly generated large prime numbers. Then, we can use this public certificate as a unique identifier, since the probability of generating a duplicate public/private keypair is extremely small [2].

  So, with such a low probability for collision, we chose to use this system for generating unique Engage Network identities. An added benefit is that we could use OpenSSL to handle key generation for us, so we did not have to write our own routines.

7.3 Network Topology

Current topology features strong connectivity in the peer to peer server network for distributing client status messages. If we imagine each server in the network as a node on a graph, then our topology is that of a complete graph on n nodes. Now, it is trivially easy to send messages to all other connected nodes by sending out $n-1$ unicast messages.

7.4 Future Plans

The first major improvement that can be made is by changing the underlying network topology. A completely connected graph of n nodes is not so bad for small n, but this structure does not scale very well. In the future, we expect there to be hundreds of individually run Engage servers. Using the current architecture, each time a client joins or leaves a server, that server must make $n-1$ copies of the join/leave message, and unicast it to each of the other servers on the network. This is a waste of bandwidth, and so we have explored other ways of structuring the network.

As previously mentioned, Chord's distributed key hashing technique seems to offer the most scalable solution to client status lookup and media distribution. Currently it is not fully implemented or suitable for use on the WAN, and so we pursue it only as a future direction.

[2] Prime numbers with hundreds of digits are used in the generation of the public and private keys. A number with on the order of hundreds of digits needs a minimum of 329 bits to be represented in base 2 (the smallest 100 digit number is 1×10^{99}, or, 1 with 99 zeros after it). Assuming the probability of a 0 or 1 is constant, and the same for all bits, the probability of choosing a duplicate bit string is at most $\frac{1}{2^{329}} \approx 9.14 \times 10^{-100}$. This is extremely small. The mass of an electron is only 9×10^{-31}kg, which hopefully puts this into perspective.

8 Outstanding Issues

Since Engage is an ongoing research project, there are always bugs we need to fix, issues that need to be resolved, and new features we wish to add.

- **JSSE does not send client certificates to the server**

  JSSE, or Java Secure Sockets Extension, is an SSL/TLS implementation developed by Sun to run on top of the existing Java network APIs. We chose to use JSSE over other SSL packages for Java because it was publically available for free, supported by Sun, and based on the existing network class hierarchy. JSSE maintains its own system wide keystores and support various certificate formats (we use SunX509). SSL sockets are created via a SSL socket factory provided in the API. All remaining operations for data send/receive are identical to the normal Socket API. Originally, we intended to use the newer TLS security API for authentication, but during the handshaking process, OpenSSL (which is used on the server) never recognized JSSE's sending of client certificates. Trying out SSLv3 resulted in successful certificate transmission, but consistent rejection by OpenSSL's verification process. We tried certificates which would successfully handshake when used with C++ clients that used OpenSSL. However, these certificates would fail when sent from JSSE's SSL implementation (which, in a similar fashion, would authenticate to JSSE SSL test servers). After investigating message board archives indicating similar OpenSSL/JSSE interaction issues, we hypothesize that this is indicative of an incompatibility between JSSE 1.0.2 and OpenSSL 0.9.6c. It remains to be seen if this has been fixed by either the JDK1.4 or later versions of OpenSSL.

- **Finish the implementation of the Engage Network**

  Due to the JSSE bug described above, we did not have a good way of testing secure SSL connections between the client and the server. So, client message passing was not implemented, and as such, the Engage Network is not finished.

- **Replace the core data structures in EngageServer with STL structures**

  There are two main data structures in the server, linked lists, and a hashtable. Both of these are homegrown implementations which have not been tested thoroughly, and we believe there are still bugs in the hashtable implementation. Replacing these structures with the STL `vector` and `hash_map` classes will significantly decrease the complexity of the code, as well as fix all outstanding issues due to bugs in our structure implementations. It will also be easier to add new features to the server in the future, since we won't keep having to adapt our linked list code to support yet another list-type class.

- **Handle more diverse media types**

  This is an issue which was not in the direct scope of our independent study, but is one that remains to be dealt with. As explained in section 2, we want to create a world where there are no clear boundaries between media types. Currently, we only support PNG files for avatars, but in the future we hope to support a wider variety of media formats. Users should be able to wear an MPEG or a sound, and the system will automatically composite their avatar to reflect the worn media (i.e. if a user wears a sound, their avatar can have a clickable button to play the sound, or the sound could be triggered when you move the mouse over the user).

- **Generalized Object model**

  We would like to support more types of objects in the world. Right now, avatars constitute one type of "object" in the world which can wear media and be manipulated and move around in the environment. What if we also made stationary objects, and scripted actions to them, and gave them their own media to wear? For example, we could make a "door" object which was a link to another room in the current environment, or a link to another Engage server. Or, we could make a URL object which could be moved around on screen, as well as visited when clicked on. All of these ideas require a more generalized object model, where the objects in the world are actual C++ classes which inherit from one common superclass.

- **Scripting support**

  It would be neat if, someday, we could create a scripting language which allowed the manipulation of world objects based on some predefined commands. For example, a ball which bounced around the screen, based on a movement script. Or, a game of chess or checkers. Scripting language support would make the Engage environment a much more interactive and enjoyable one.

9 What we learned

- **Java UI programming with Swing**

  The Java client's Swing interface presented a departure from most CS programming assignments, which are command line based. The project introduced us to Java's frame based UI structure, support for various interface widgets, and primitive drawing commands.

- **Front/back end network code design**

  Network applications that rely on a GUI interface have to meet the responsiveness demands of the user while still processing incoming network messages at any time. We learned methods of meeting both requirements by separating UI and network tasks on separate, preemptive threads. This division of labor is most evident in our UI network activity callback, which always puts UI work generated by network events on the UI thread through the use of the SwingUtilities.invokeLater() method.

- **Byte packing with endian support**

  To support our legacy code which didn't communicate properly on different hosts because of endian differences, we developed an all purpose byte ordering class which can pack and unpack all relevant primitive types from byte arrays. This was handy because Java's network traffic is all received as raw byte arrays. Simply setting a flag indicates whether to unpack little or big endian data.

- **SSL theory and protocol structure**

  In order to achieve secure communications, we used public SSL libraries for C++ and Java. We researched both the underlying SSL protocol and implementation details with JSSE and OpenSSL. Our poster from the Meeting of the Minds conference has a diagram of how an SSL handshake is performed, and is shown in figures 2 and 3.

- **Theoretically efficient means of P2P file distribution**

  As explained in section 3, the Chord project from MIT can be used to distribute state among different servers. Justin is also convinced that Mercury can be used to create a massively scalable, shared environment, but this would defeat the purpose of keeping different Engage servers segregated based on owner.

- **Source code management**

  One issue we had when first working on this project was that of managing our source code. During our initial development stage, we ended up sending each other compressed archives of our source, and somehow managed to keep things separate enough such that this was a relatively painless process. But, we always wanted to be able to merge what we were doing, and share the work in development. So, as one minor subgoal of this project, we learned how to use the Concurrent Versions System (CVS) to manage our source code. Learning CVS actually proved to be worthwhile, since we have gone on to use it with other projects and in our other classes.

- **Cross-platform programming**

  Since one of our goals was to port the Engage Server to Linux, we needed some way of managing the source code between the two operating systems. We decided that the best way of doing this was to keep using Project Builder for OS X, and create a separate Makefile for Linux. Then, during the compilation process, a symbol would be defined depending on what OS the source was being compiled on; for OS X it was "DARWIN", and for Linux it was "LINUX". This amounted to invoking `gcc` with the `-DDARWIN` or `-DLINUX` flag. Then, in the code base, whenever there needed to be a distinction between code specific to each target OS, we could wrap it around an `#ifdef` statement. One example of where this is needed is with the `accept` function, which requires slightly different arguments on each operating system.

The Engage Communications System

Justin Weisz and Michael Piatek, SCS
Advisor: Dr. Srinivasan Seshan, SCS

Abstract

In the course of using any type of communications software, groups of users tend to coalesce, and meet each other regularly on one server. The problem with this cohesion is that when one or several members of the group decide to explore other servers, those members of the group are unable to communicate with the other members until they return to the server they left. This inability to communicate across servers leads to major fragmentation of user groups, which is the problem we wish to solve.

What we propose with the Engage Communications System is a method for allowing groups of people to communicate simultaneously, while preventing the group fragmentation problem from occurring. This is done by forming a network of individually run Engage servers, where the state of each server is exchanged. Clients can be notified when their peers join other servers on the network, allowing them to communicate directly.

Network Organization

• The Engage Network is organized as a complete graph on n nodes. This allows us to easily test message passing, while ignoring other issues such as routing and more complex network topologies.

• The protocol we developed includes a message type identifier, which allows us to handle and support a wide variety of messages, and provides us room to grow for future message types. Current message categories include: media messages, chat messages, network messages, and user join/leave messages.

• In order to ensure security, as well as a unique method for identifying clients across servers, we employ OpenSSL, an open source implementation of the Secure Sockets Layer (SSLv3) and Transport Security Layer (TLSv1) protocols. SSL encrypts the messages sent between clients and servers, and provides digital certificates, which uniquely identify each client on the network.

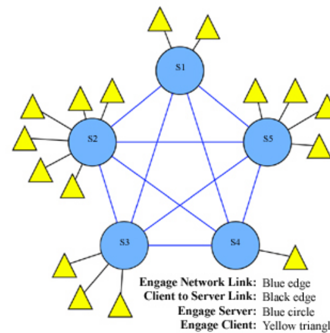
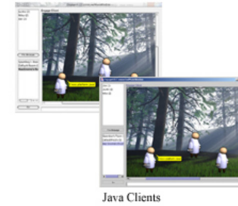


Figure 2: Our Meeting of the Minds Poster - Left Side

Engage Client

The Engage client showcases the pervasiveness of the graphical metaphor in our communication system. Our Java client ensures cross platform compatibility while our OS X client demonstrates the feasibility of a natively hosted client. Native hosting allows us to take advantage of services not offered in the Java API such as text-to-speech. The Java client source code includes a network layer for interacting with network messages independent of the user interface. This separation of services makes a native interface possible without rewriting network code that doesn't affect the user experience.



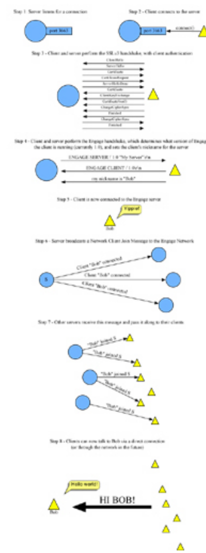
Java Clients

Engage Server

Engage Server (Mac OS X hosted)

The Engage server is the heart of the network structure. It is a POSIX compatible, command line tool intended to handle the routing of both messages between clients connected to just one server and messages between servers on the server peer network. The server peer network maintains redundant lists of online users so each server can route messages to the server where their target is connected.

Servers also employ certificate-based authentication to verify the identity of every user on the network (via SSL/TLS). To this end, some servers may act as certificate authorities that listen for special connection requests. When received, the server signs client certificates using the Engage Network master key. The client or server bearing this certificate is now a trusted member of the Engage Network, and can communicate with other members who have had their keys signed by the same authority.



Future Directions

Application of distributed hash lookup systems for server server message routing such as Chord from MIT (http://www.pdos.lcs.mit.edu/chord/). This system analyzes an identification number in order to locate a file using O(log n) messages. We can apply this concept to make finding a user more efficient. Currently, redundant active user lists are stored completely on each server node and user login and logout messages must be repeated to all nodes. Chord presents a method of dividing the list storage space and decreasing network traffic.

Enhanced support of media in the client. Currently, only graphical representations of users are permitted. Ideally, we would like to represent any file generically as a world object.

Figure 3: Our Meeting of the Minds Poster - Right Side