**Team 17 Methodology, Anonychat**

Ben Korza, Mark Fitzgibbon, Mitchell Caisse

The Anonychat program will consist of a client/server program to be used by individual users, as well as a centralized name server to confirm peers to connect are online. The project will be written in C, for Linux Operating Systems. Other operating systems are not a focus for our project. We will implement our chat program using a distributed client, in which clients will broadcast their messages to their peers. The messages will also be encrypted; this will allow for client messages to remain anonymous and their contents secure.

In order to manage adding users to an Anonychat network, a central name server will manage connections between clients. The name server will be responsible for keeping track of what clients are available, what to do when clients join and leave the network, reassigning connections, and guaranteeing that no clients become isolated from any other client. The central name server will not be responsible for routing messages between clients and is to be excluded from the list of possible client connections when they are assigned. The central name server will also make no attempt to identify clients connected to it beyond IP and port address, data which any program would transmit during a normal connection, and therefore should not harm our attempts at maintaining anonymity. The name server should also be capable of handling connections from proxy’s or behind NAT for the same reason.

Every client starting will initiate a TCP connection to the central name server, and wait for a response. Since the connection is continuous, the name server will know when a client loses connection, and can react accordingly. The name server will maintain a graph of all connections within the network, and will reform the graph regularly. A cycle for forming the graph will occur as such:

1. An internal list of clients is re-arranged to occur in a random order. This should ensure that connections appear random as well for the following steps.
2. Based on a series of internal parameters, such as minimum number of connections and availability of clients, nodes are arranged so that they connect, bidirectionally, into a ring, with an arrangement of connections occurring to cross the ring. If the requested minimum number of connections would mean that some connections would overlap, every node will connect to each node once. Our testing will start with a minimum number of 5 connections, and can be adjusted.
3. When the central server’s graph is completed, a message is sent to every node that informs it of only where it is allowed to broadcast to. No node is informed of who is assigned to send to it, and the bidirectionality of the connections is only implied by the specific graphing implementation used. Future name servers can be made with differing graphing mechanisms. Additionally, nodes are not to be informed in any way of connections beyond one hop. As a result, attempts to send to a node that is no longer available will simply die in the network, possibly never reaching the target. This is by design, as it means that if no one has the requisite keys to read a particular message, then the message eventually fades.
4. This cycle will repeat either (1) every time a client joins or leaves if the number of clients in the network is less than a set number, or (2) every 30 seconds if there are a larger number of clients. The second case implies that, in larger networks, new clients may have to wait up to 30 seconds before it is able to broadcast messages, but this should also keep the name server from flooding the network. Messages in transit when connections are re-assigned behave as if nothing happened.

A client needs two things to communicate with others, first is the list of peers discussed above, second is public and private keys. We have decided that providing a platform to exchange and generate public/private keys is out of the scope of our project, and users will be responsible for generating their own public/private keys and distributing the public key to whom they wish to communicate with.

Once clients have a list of peers and the public key of the user they wish to talk to, they can follow the steps below to send a message to them.

If Alice wants to send a message to Bob, Alice’s client will:

1. Craft a message.
2. Encrypt it using Bob's public key.
3. Send the message to all of its known peers.

The N peers receiving a message will need to determine if the encoded message they received was intended for them to read or to be sent out and if they have seen the message before. When a message is received, a client will make a hash of the received message, and check against an internal list of hashes. If the hash does exist, then the message is not new to the client and nothing will be done, since it has already been processed. This will reduced network usage by not having a client forward a message if it has already seen it, making the chance of a cycle zero. If the hash proves to be new, it is added to the list, along with a time to live, and the client will decide if the message was intended for itself or to forward it on. To determine if the message was meant for it, the client will attempt to decrypt it with all of its private keys.

A successful decryption, determined by a sentinel string in the message, will imply that the client is meant to display the message to the user. If the message was unable to be decrypted, it will be passed along to the clients set of peers. The specific number of peers in broadcast range will need to be determined when a message is being sent based on what the client knows exists.

The message protocol that will be used for communication between clients as well as clients and the name server can be seen below. This protocol is based upon the IRC protocol, and will have a maximum length of 512 bytes. It is terminated by CRLF, which is included in the message size. It will also be prefixed with a sentinel (encryption prefix) when being sent between clients, the name server will not have one.

ENCRYPTION\_PREFIX [<prefix>]<command> <params> <...> <crlf>

Thus far, our implementation is still vulnerable to an adversary that has a view of the entire network as a whole. This is because a theoretical adversary with this power would be able identify the first broadcast of a message, and from that determine who sent it. In order to solve this problem, we turn to an approach similar to onion routing. In onion routing, messages are encrypted with multiple layers of encryption and decrypted layer by layer as the message is bounced around the network though different onion routers. Inspired by this, we use a simpler approach. As previously stated, all the users share their public keys when they connect to the network, and anytime a user sends a message, it is encrypted with the user’s public key. However, as the message is sent between nodes, each node also encrypts the message with the next hop’s public key. This constitutes a second layer of encryption, with the end effect being similar to onion routing; it makes it harder for an external omnipresent observer to trace message senders in the network.

To test the success of our protocol, we plan on running multiple tests with different network conditions. To test latency, we plan on running our clients on four different machines outside of each other’s local network. Four is the maximum number of clients we could achieve practically for this test. We will record the latency figures by using printouts displayed before a message is sent and after it is received.

We also plan on conducting a second test that will run 15 of our clients on the same local network. This test will be used to test the scalability of our protocol; that is, whether our implementation can continue to send and receive messages in a timely manner over a larger network, as well as if the name server can handle mapping these clients and whether the packet flooding remains under control. In the course of this experiment we will also use wireshark to try and intercept messages and confirm their encryption status.