**Team 17 Methodology, Anonychat**

Ben Korza, Mark Fitzgibbon, Mitchell Caisse

In order to accomplish our goal we have designed a messaging protocol that will provide anonymity through the use of a distributed network of peers and multiple layers of encryption. In order to test our design we have implemented a test platform in C for Linux. This test application will allow us to perform testing on our design to ensure that it meets our goals and does not have a huge performance overhead. The details of our design and testing methods will be discussed in detail below.

The Anonychat messaging protocol will consist of a set of distributed clients to act as peers in transmitting messages, as well as a centralized name server to inform clients of available peers. These clients will send messages amongst each other using controlled flooding, encryption between the source and destination peers, and encryption between each intermediate peer. In order for this to work, there we needed a way to ensure that each peer has a connection to all of the other peers. This is the role of our name server, it will distribute peer lists to each peer, and make sure they can all communicate.

The Anonychat Name Server (ANS) is responsible for creating a network of peers that provides each peer with a route to each other. Our protocol does not require a specific implementation for creating this network, but there are certain conditions that must remain true. The peer network must be changed periodically allowing peers to send messages on different paths. (The peer network must assign peers in a random fashion, for example peers with similar IP address prefixes, or latency should not be assigned to each other.) As stated above, the peer network must not isolate any peers or group of peers, they must all have a route to each other. The ANS must also maintain connection state on all peers currently on the network, and be able to react accordingly when a peer drops from the network.

In our test implementation we maintained state by requiring each peer to maintain a TCP connection with the server. This allowed us to determine when a peer has connected/disconnected and also send out peer updates to the peer when required. Our node network was implemented using a connection graph, which is detailed below:

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 have a bidirectional connection, 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.
3. When the central server’s graph is completed, a message is sent to every node that informs it of its assigned peers. 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.

The client component or peer, of our messaging platform is slightly more complicated and requires two levels of encryption in order to maintain anonymity and confidentiality as well as controlled flooding to communicate the messages. Both levels of encryption will use asymmetric encryption, implemented by the RSA algorithm. The first level of encryption, between the destination and source, guarantees that the destination knows the message is for itself, without the source knowing any information about it other than its public key. The second level of encryption ensures that the source and destination of the messages remain anonymous. This level is between each intermediate peer, and ensures that the signature of the message changes as it is sent across the network, but the content of the message remains the same, providing source anonymity.

Peers will communicate with each other by broadcasting the message to all of their peers, and their peers will also broadcast the message to their peers and so on, which is also known as controlled flooding. When a peer receives a message that it has never seen before, it will always broadcast the message to its peers, regardless of whether it was the intended destination or not. To determine if a received message is new, a hash of the source/destination encrypted message can be compared against a hash of previously received messages.

The message protocol that will be used by our messaging platform is relatively simple. It consists of a ten character or less command with the body following, delimited by a space. All messages will have a fixed size, to further ensure anonymity, making a message unidentifiable by its size. In our implementation our encryption method always results in messages of equal size, creating fixed sized packets, at both layers of encryption.

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 alleviate 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, which is generated on connection and destroyed on disconnect. 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 verify that our protocol is viable, has sufficient performance, and works as intended, we plan to test message latency, latency introduced by encryption, time difference in successful and unsuccessful decryption, and our controlled flooding implementation does not have a huge network utilization. To test these we plan to run a series of simulations on a local machine and four computers spread out on different networks to get accurate latency results.

We expect that our protocol will add additional round trip time to our messages as they are send indirectly, but we don’t want this additional latency to be much bigger than the latency between direct communications. We will set up our test implementation on our four test machines. We will then send a series of 25 messages between each machine using our messaging protocol, and record the round trip time of each message. We will then send another 25 messages between the each of the four hosts themselves using ping, and record the round trip time of each message. After collecting the round trip time of both messages, we will compare them and verify that the RTT using our protocol is not greater than five times the direct RTT.

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, as it creates a reasonably sized network to view packets. 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.