Peer to Peer Project

Jan H. Knudsen (20092926) Roland L. Pedersen (20092817)
Kris V. Ebbesen (20094539)

June 1, 2014

Contents

1 Introduction

With the recent rise in awareness about the blanket surveillance of our daily internet usage, the general population no longer feel that the anonymity and security of their online exchanges are guaranteed. We hope to amend this.

In this project report, we describe how to develop a peer to peer system that allows the exchange of encrypted chat messages, while making it quite difficult to determine who sent and received what messages. The system is built on top of an existing unstructured network, and the techniques we use could be applied to many different unstructured networks. This means, that with minor extensions, existing networks can be allowed to support this kind of traffic. We will describe both why we have chosen to develop this system, how it works, how to operate it, and show experiments that measure the performance of the network.

2 Related Work

Protecting freedom of speech and the free exchange of ideas in the face of government oppression and censorship is one of the nobler goals of P2P networks and cryptography. Several systems and protocols with this as their main goal have been designed and analysed over the years. One such project is Freenet (Clarke et al., 2002) which builds a virtual file space distributed among peers and provides censorship resistance by being anonymous, decentralized, encrypted with peers only knowing about direct neighbours, and providing peers plausible deniability by having the shared resources be encrypted so forwarding peers can't know the content. Freenet, however, provides very low performance and no guarantee that a file will remain, as replication relies on the resource being requested by other peers and other peers choosing to cache it along the way. Resources are also available to anyone, meaning encryption of shared resources and another channel to distribute the keys is paramount. Resources also cannot be sent to a specific target.

A different system that provides something closer to a real-time instant message client is BitMessage(Warren, 2012). This, however, uses flooding to send messages across the network and as we have demonstrated in our previous paper (Ebbesen et al., 2014), there are better performing alternatives, such as random k-walkers, and we would like to see if we could implement something as secure as BitMessage using these different methods of sending messages.

3 Use Case

When developing the system we maintained clear goals for the required features of the system, which helped us not only remember why we were developing the system, but also guide the project in the right direction.

These goals are best illustrated in use case format, as follows:

3.1 Use case 1: Chinese Dissidents

Primary Actor: 2 Chinese citizens with a wish for democracy.

Goal: To exchange thoughts and literature about democracy in a way that is safe, secure, and not under government scrutiny.

Other interested Parties: The Chinese government.

Preconditions: It is assumed that both citizens have exchanged either their full public keys or a shorter identifying string prior to the use case (by carrier pigeon). In order to avoid data analysis attempts, a secure channel to a peer located outside the reach of the Chinese government would be also preferred.

Success Criteria: Using the developed system, the two citizens should be able to send chat messages to each other, with a guarantee that they cannot be read by 3rd party actors. Additionally, they should be able to know who sent the messages, and whether their messages reach their destination. Finally, it should be difficult to determine that these two citizens are communicating.

Method:
1. Both citizens add their private key to the system.
2. Both citizens set up the public key of each other in the system.
3. The citizens start sending messages.
4. The citizens look for confirmation messages from the system.

3.2 Use case 2: Young Love

Primary Actor: The young doe-eyed girl Earlene.

Goal: Earlene wishes to profess her love to Billy Ray, the handsome new farmhand, in intimate prose, yet she does not wish to reveal her identity yet.

Other interested Parties: Billy Ray, the manliest farmhand around.

Preconditions: It is assumed that Earlene has obtained a hash address of Billy Ray, either through teenage girl gossip, or a phone book. Also required is Billy Ray's willing publishing of his public key.

Success Criteria: Earlene should be able to express her innermost desires to Billy Ray, without him know her identity.

Method:
1. Earlene uses the system to fetch the public-key of Billy Ray.
2. Earlene condenses her desires into a 20-page novella with graphic descriptions of the surrounding landscape.
3. Earlene sends the message unsigned to Billy Ray.
4. The message is received at Billy Rays end, and Earlene receives a signed acknowledgement.

Additional details: Should Earlene wish to continue secret communication with Billy Ray, she might generate a new RSA key pair, publish the public key, and attach the corresponding hash address in her message.

4 A short note on cryptography

The project described in this report relies heavily on cryptographic system and practices in order to be possible. It is however the case that none of the persons working on the project have any previous experiences working with secure systems, since we come from backgrounds in algorithms and computer graphics. We have chosen such a heavy reliance on systems outside our normal line of work as a learning experience, and with the strong conviction that most cryptographic systems work well as black boxes.

Expanding further on this, when we refer to a cryptographic method, we will rarely expand on the inner workings of such components, but rather rely on their security as provided by their developers. Of course this makes us somewhat prone to making errors that would be considered mistakes by hardened security veterans, but we beg forgiveness for our bright-eyed naïveté.

In the end, what matters to us in this project, is the parts that relate to peer-to-peer systems.

5 Method of Operation

5.1 Base System

The system described is built atop the unstructured network developed during the P2PN course (Ebbesen et al., 2014). This network contains very little structural information, and bases its topology on the GIA network (Chawathe et al., 2003).

The choice of this network was made based on its simplicity and extendibility, and due to the fact that unstructured networks require little information about the peers involved, making it difficult to track which peers are doing what.

To make the system guarantee an eventual delivery of the message if the sender remains connected, we extended the k-walker algorithm to work somewhat akin to the expanding ring search algorithm wherein the TTL value is increased and the search repeated when a search fails. We send out a number of walkers and wait for a while, if no acknowledgement has been received after the wait, another set of walkers are sent with double the TTL and double the wait time. Starting values of TTL and wait time is configurable, and good values will be found through experimentation and measuring.

Note that the techniques used to extend the network could be applied to most unstructured networks, and would probably work just as well on the GIA network.

5.2 Encrypting Messages and Hiding Recipients

In order to ensure that no adversaries can read the content of any given message, we encrypt chat messages travelling across the network using RSA-OAEP (Bellare and Rogaway, 1995). RSA-OAEP was chosen due to its ease of use, and security against repeated plaintext attacks.

When performing this encryption, we use a pair of RSA keys. The sender must obtain the public key of the final recipient (how to do this will be explained later), in order to encrypt the message.

When the chat message is sent, it is first encrypted by RSA-OAEP using the public key of the recipient, and then broadcast across the network using either flooding or k-walkers. Whenever a peer receives a messages travelling across the network, it will attempt to decrypt it using the corresponding RSA-OAEP decryption using its own private key. This will fail for all peers except the recipient, ensuring that only the final recipient will be able to obtain the contents of the chat message.

Note that the encrypted message sent across the network contains no delivery address of any kind, and as such no other peers will know the final recipient.

It is also worth noting that only one RSA key pair is required to send messages. The sender needs no private key, nor do any other peers in the network except the receiver.

5.3 Signed Messages and Acknowledgement

All chat messages in the system may or may not be signed by the sender. If the sender wishes not to sign his messages, in order to hide his identity from the receiver, or because he is not in possession of a private key, he may omit this signature. Additionally, any message received by a peer can be acknowledged by returning a signed digest of the message.

Both types of signatures are done according to **PKCS#1! v1.5** (Jonsson and Kaliski, 2003).

In the case of the sender signing a message, we send a signature of the plain-text message along with the encrypted message. This ensures it will only be possible to verify the signature after obtaining the decrypted message, and that we keep the identity of the sender hidden to anyone except the recipient, and that the recipient can securely verify the sender given his public key.

When verifying the delivery of a message the receiver returns a signed digest of the plaintext message, which is verified by the sender. This ensures that the sender has received the message, as he is the only one able to provide a valid signature. If the peer is using flooding we simply return this value as part of the xml-rpc call, while we answer back using a k-walker in the case that we receive a message by k-walker. Given a small random delay, it becomes difficult to determine whether a message was received by any given peer, or one of his neighbours.

It should be noted, that if a message content is tampered with before being delivered, the signature of the sender will no longer be valid. This ensures that any message that is tampered with will have no valid signature, and be seen as an anonymous message. Additionally the receiver signature for message delivery guarantee will not match the expected signature at the sender's end, and the message will not report it as being delivered.

5.4 Encrypting Peer Communication

All traffic between peers in the network is encrypted using anonymous Diffie-Hellman (Diffie and Hellman, 1976) encryption. This encryption is provided by wrapping connections between peers in an SSL layer, with no certificates and anonymous Diffie-Hellman as the only cipher set.

This ensures that peers can communicate without outside parties snooping on the information, which makes it very hard to track messages across the network, since the data sent from messages, cover traffic, and general networks operations will be indistinguishable.

Another reason to use anonymous Diffie-Hellman encryption is that it enforces no requirements on previously distributed keys or identities of the peers, keeping each peer's knowledge about its neighbours at a minimum.

In order to prevent constant Diffie-Hellman key renegotiations we provide cached pools of SSL connections, meaning that we only create a new connection when the peer runs out of idle connections to the same peer.

5.5 Cover Traffic

Inspired by the use of cover traffic in the Tarzan p2p protocol (Freedman and Morris, 2002), we include cover traffic in our solution to prevent traffic analysis. The network relies heavily on the SSL encryption of the peer-to-peer connections to keep traffic types indistinguishable, making it very difficult for an outside observer to discern what data traffic belongs to messages and which concern the network.

In terms of cover traffic, we provide two sources of cover.

One is the general operations of the network. Neighbours will constantly contact each other to ensure that they are alive, and any peers leaving or joining the network will require a fair bit of communication between peers. Since all of this traffic is encrypted, it will hard to distinguish this communication from messages.

The second source of cover traffic is explicit cover traffic. Peers will at random intervals send random data of random lengths between each other. This data ensures unpredictable network traffic, and hinders traffic analysis even further.

5.6 Providing Proof of Work

To keep the network stable, and free from Sybil-style attacks, we use a proof of work system. This ensures that peers that wish to put a strain on the network, or affect the overlay network structure, will need to expend large amounts of computational resources to do so.

The proof of work system is based on HashCash (Back, 2002), and requires a peer to generate a partial hash collision with the timestamped resource, using the SHA-256 hashing algorithm. How large a collision and how new a time stamp must be is fully configurable.

A proof of work is currently required in 2 circumstances. The first is when a peer wishes to join the neighbourhood of another peer. In requiring a proof of work for joining or moving within the network, we make Sybil and Eclipse attacks less effective, while imposing little to no hindrance on long-term stable peers. The second proof of work is required when a peer wishes to send a message. This is to deter spamming of the network, and to prevent malicious peers from forcing other peers to spend an unwanted amount of time trying to decrypt messages, or drown a single peer in messages after having obtained its public key. We have used either few or no required bits of a proof of

work in our testing, unless specifically testing the proof of work, in order to allow rapid testing of the network.

5.7 Public Key Distribution

When a peer wishes to communicate chat messages to another peer, it is required to know the public key of the recipient. This public key can be supplied directly by the sender, indicating that the key has been distributed securely outside of the network. In this case, the key is simply loaded from a provided file.

The network also offers the option of publishing public keys using the underlying peer-to-peer networks ability to share resources. When doing this, the public key is read, and stored in the network as a resource using the base64 encoding of its SHA-256 hash as its name. The key can then either be fetched and stored normally as a resource by other peers, or loaded directly into the public storage of other peers. Any peer that loads the key directly will verify its hash as it does so. The result is a tag (44 characters long), that can be shared much easier than an entire public key.

5.8 Limiting RPC Call Availability

Standard practice in object-oriented Python-based RPC servers is to register the entire object for RPC call availability. This is highly inadvisable if one wishes to protect the network from malicious peers.

In order to prevent this form of attack, we enforce strict limitations of function availability. This is done by extending the way the RPC calls are handled by the XML-RPC components, and tagging only the needed methods calls as being callable by RPC. Any attempt to call an unlisted function will silently be ignored.

6 Experiments

We have performed several experiments on our system, both to show that our security measures work, to find optimal parameters and to compare flooding and k-walker approaches with our added security features.

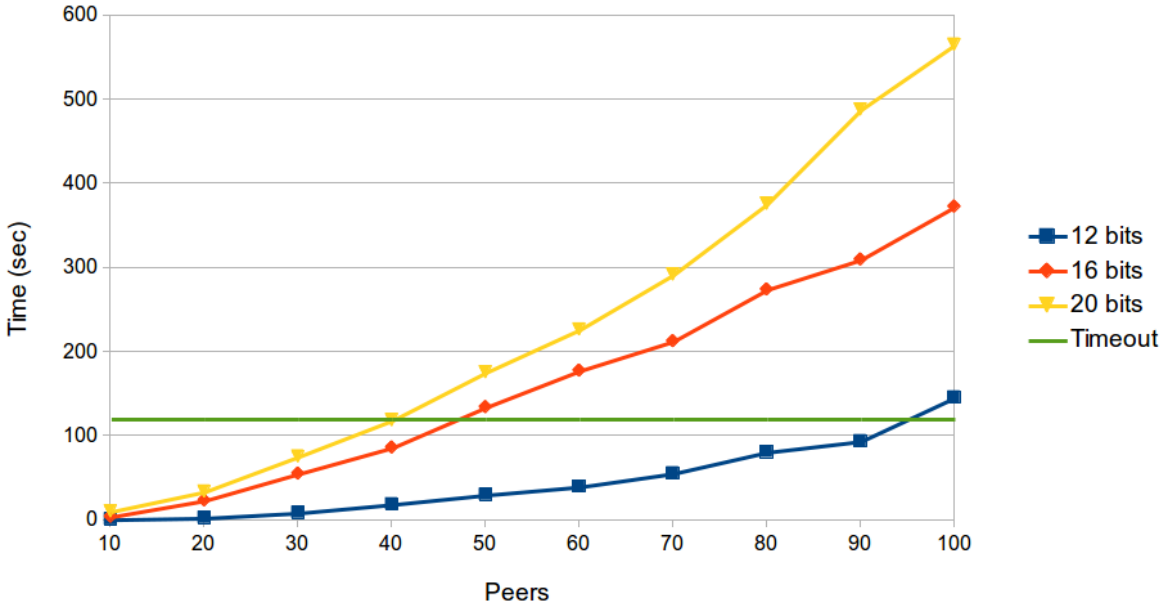
6.1 Proof-of-Work hinders Sybil Attacks

We prevent sybil attacks by enforcing a strict requirement of proof of work on peers wishing to join the neighbourhood of a healthy peer. It is quite hard to test the actual time required to perform such a task, but we did a simple test of measuring how quickly we are able to create a sybil-like group of peers, and make them appear as a real network, while adjusting the number of bits required in the partial hash collision.

Note that our network enforces a 120 second maximum limit on the age of such proofs of work, to further limit the effectiveness of sybil attacks.

We see in Figure 1 that with a proof of work requirement of 20 bits, it becomes infeasible to create much more than 40 peers, while more than 90 peers can effectively be created at 12 bits. In reality, one might wish to make the number of bits even higher.

Figure 1: Proof of Work



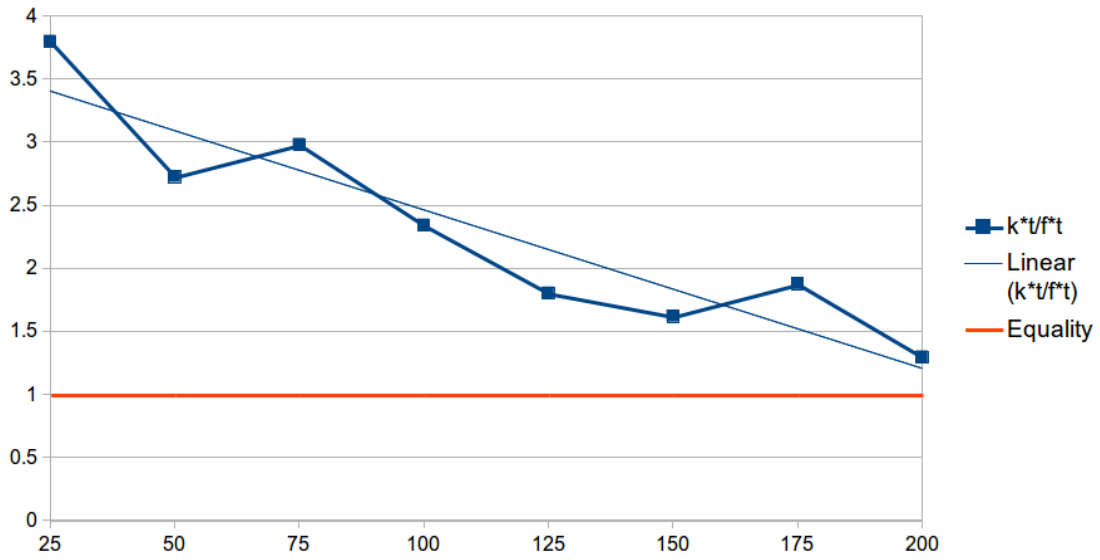
It should be noted, that theoretically, the amount of time required should grow by 16 each time we increase the number of bits by 4. The reason our test shows nowhere near this huge increase is likely that the peers simply get more starved for connections, and form a much sparser network. This means that it is still possible to connect quite a few sybils to the network, but they will have a lesser effect on the other peers, due to their low connectivity.

6.2 Finding optimal k-walker parameters

As described in section 5.1, the k-walker algorithm implements an eventual delivery guarantee as long as the sender remains connected to the network by sending out walkers with exponentially increasing TTL at exponentially increasing intervals. Before doing the experiments comparing flooding and k-walkers we needed to find some optimal or at least pretty good parameters for this.

The required parameters were the starting values of wait time and TTL as well as how many walkers to send in parallel each time. Tested values of starting wait time include 0.5, 1, 2, 4, 8 and tested starting values of TTL include 8, 16, 32, 64, 128 and the tested numbers of walkers to send out time include 1, 2, 4, 8, 16, 32. Since each test took quite a while and had to be repeated several times to try different network layouts, we didn't test all possible combinations of the values but used our intuition and understanding of the effect of the values to zero in on some good values to test further. We found that the network layout had a huge effect on the time and number of passed messages in the network to send a single message and receive verification of delivery. We omit presenting the data from this experiments as not even close to every combination of the parameters were tested, making graphing the values useless, and not every test was written down.

Figure 2: K-Walker / Flooding: Time



We ended up using a starting wait time of 4 seconds and starting TTL of 64 and send out 16 walkers in parallel.

6.3 Flooding vs. k-walkers

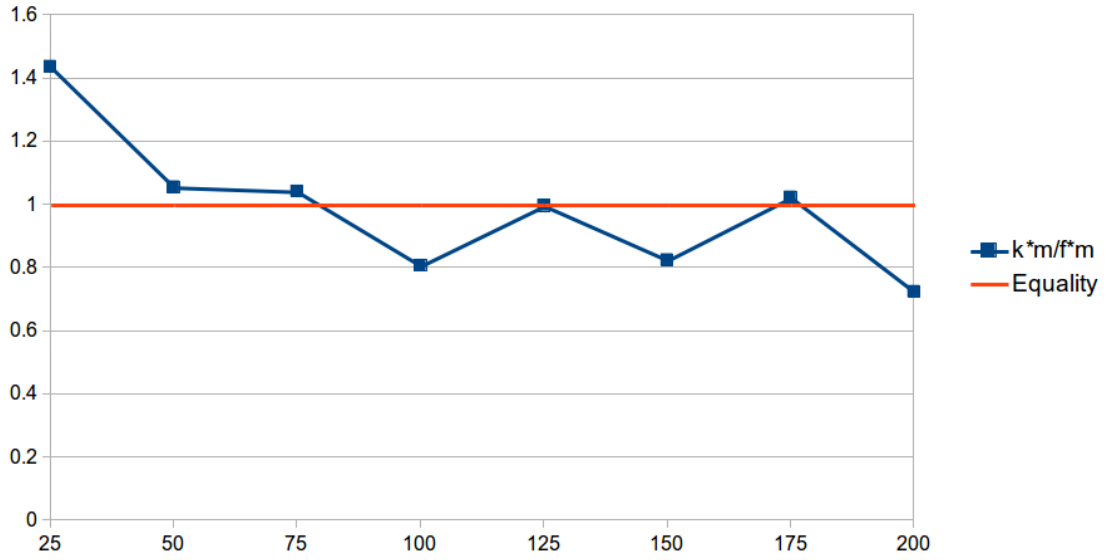
We tested the time and internal messages passed in the whole network by having two peers join a network of varying size and sending 10 messages from one to another, waiting for the acknowledgement of the previous message before sending the next. This test was then repeated to test various network layouts. These tests were performed with Proof-of-Work turned off and no artificial latency introduced in the system in network of size increments of 25 peers.

We weren't able to have networks involving more than 230 or so peers running on the same machine without random peers dying or at least not responding fast enough so we limit our testing to a max network size of 200.

In Figure 2 we have graphed how the k-walker algorithm performs compared to flooding by dividing the time of the k-walker algorithm with time of the flooding algorithm run on the same network, then averaging this over 5 runs on different network of same size for a value in the final graph. The same was done for Figure 3, except the number of messages for k-walkers and flooding was used instead.

We see in Figure 2 that k-walker in general takes more time than flooding, but also that its relative performance increases noticeably as network size grows. This suggests that the k-walker likely will outperform flooding in larger networks and, looking at the trend line, even in networks not much larger than our test sizes, perhaps at about 225 peers. In a realistic setting such as one of our use cases, we expect the network to reach much larger sizes, perhaps in the thousands, and so we expect a k-walker algorithm to

Figure 3: K-Walker / Flooding: Messages



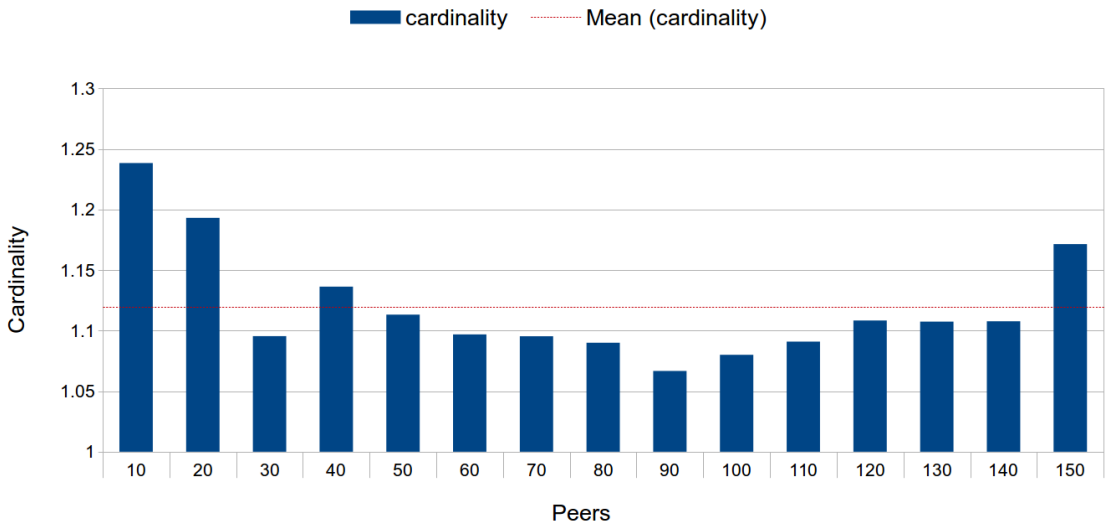
perform much better than flooding time-wise in real world use.

In Figure 3 we see that the k-walker algorithm in general either outperforms or performs as well as flooding in regards to number of messages when ignoring very small network sizes. If we had used different parameters for the k-walker algorithm, such as sending fewer walkers in parallel, we find it likely that k-walker would also have outperformed flooding at small network sizes. different parameters would also likely have been able to make the k-walker algorithm outperform flooding even more in larger networks.

The graph seems to oscillate, only barely showing a trend towards better performance at larger networks. We explain this by the fact that the number of messages, as well as time, fluctuated greatly in our tests depending on the configuration of the exact network it was running on. It is also sad that we could not test on a network of a greater size than about 200 peers, as we expect the k-walker to be better at larger networks and we are unable to show it.

It should be noted that the k-walker algorithm waits a full 4 seconds after receiving no verification of delivery before sending out more walkers, and this will artificially inflate the time it takes on smaller networks where the wait time could easily be shorter as a verification walker will have a smaller network to traverse. Another disadvantage for the k-walker algorithm in this test, that also affects the amount of messages, is that it has a number of parameters which cannot be optimal for every network and network size, and we only spent a limited time trying to find some good parameters as can be seen in section 6.2.

Figure 4: Cardinality



6.4 Cardinality of Diffie-Hellman Pools

When working with the Pooled Diffie-Hellman connections, we wished to ensure that we made as few Diffie-Hellman negotions as possible. In order to measure this performance parameter, we started a network, and gave it time to settle down, while performing cover traffic. We then measured the average cardinality of the connection pools residing in the peers. This number should preferably be close to 1.

The test was done specifically with 100ms faked latency between peers, and by waiting 2 minutes while cover traffic and network kept generating data. We ran the test for 10 to 100 peers, in increments of 10. A higher number of peers was deemed unnecessary, as peers seemed to have little influence on the cardinality of the connection pools.

From this test, in Figure 4, we see that we have a cardinality ranging between 1.07 and 1.25, which means that we are generating 7-25% additional connections. The average amount of connection overshooting is very close to 12% which we deem to be good. Note that the cardinality does not appear to rise with an increasing number of peers, and in fact the highest numbers seem to reside at very low peer numbers. We theorise that the high cardinality of low peer numbers might be caused by a greater amount of loops in the RPC call graph, leading to a forced additional connection.

7 Manual of Operations

In order to operate a peer in the network, one must rely on either python scripting against the peer class of the source code, or the supplied command line interface.

We here explain the commands required to operate the peer using the command line interface. Scripting directly against peer class is left as an exercise for the reader.

Note that the peer still supports most of the commands of the original network (Ebbesen et al., 2014).

The commands are as follows:

hello [***address***] Attempt to join the network. An optional address parameter may be specified in order to bootstrap against a known peer.

After joining the network, the peer will be ready to add keys, and chat. Please note that it might take several seconds for the peer to establish an acceptable amount of neighbours.

secret ***private_key*** Load the given private key from a local file, and set is as the current key used for decrypting and signing messages. This is required in order to receive messages encrypted with the corresponding public key, and to sign messages sent from the local peer. Note that the key must be a RSA private key in the *pem* format.

friend ***name public_key*** Load the given public key from a local file, and associate it with the alias provided by the name parameter. This is required in order to send messages to the peer with the corresponding private key, and to identify that peer as a sender. Note that the key must be a RSA public key in the *pem* format.

publish ***public_key*** Make the given public key available for retrieval through the peer to peer network. Shortly after entering this command the peer will display a hash of your key, which you can share. This allows other peers to download your public key through the network if they have the corresponding hash string. Note that the key must be a RSA public key in the *pem* format.

friend ***name hash*** Fetches the key stored in the network under the given hash, and checks for hash validity of the key. If successful, the public key retrieved will be associated with the alias specified in the name parameter.

message ***name message*** Attempts to deliver a message to a friend added under the alias specified by the name parameter, with the content of the message parameter, using flooding. The message will be signed if possible, and a report of delivery given.

kmessage ***name message*** Attempts to deliver a message to a friend added under the alias specified by the name parameter, with the content of the message parameter, using k-walkers. The message will be signed if possible, and an id will be given, which allows matching to a later received acknowledgement.

8 Known Vulnerabilities

In this sections we describe areas where we know our system to be insecure. Some of these are lack of functionality in the system, and some are fundamental problems in the underlying components. Note that we will not mention attacks such as cracking RSA-keys, since these attacks are very general, and much more far-reaching than our system itself.

8.1 Anonymous Diffie-Hellman Man in the Middle

Encryption between peers is based on the Anonymous Diffie-Hellman key exchange scheme. This scheme does however have a major security flaw, in that it is wide open to man in the middle attacks. Unfortunately, since we do not have any prior knowledge about our peers before initiating the encrypted connection, it becomes quite problematic to use any of the more secure transport encryption schemes.

8.2 Eclipse-based traffic analysis

If an adversary can completely surround a peer, it becomes quite easy to determine when that peer has sent or received a message, since it is possible to monitor which messages and acknowledgements exit the peer without having come in. Proof of work for network relocation limits this, but it does not make it impossible.

8.3 Python XML-RPC is Insecure

The XMLRPC library distributed with the python package is vulnerable to several types of Denial of Service attacks. Most of these are quite applicable to most XML-based system. More information can be found at (Foundation, 2014).

8.4 Denial of Service on Key Distribution

Currently, the key fetching mechanism relies of the underlying network having a reliable way of retrieving resources. In our current system, a single malicious peer could report itself as the location of any requested resource, and respond with garbage data when asked for its value. Note that this can not be used to swap public keys, only prevent their distribution.

8.5 Key Hash Collision

If it proves possible to generate RSA key pairs where the public key has a specified SHA-256 hash, it is easy to fool the key distribution mechanism. We do however not know of any attacks of this kind, and believe it to be quite difficult. Crypto people might prove us wrong.

8.6 Spamming without Proof of Work

Currently, we require proof of work for sending messages and requesting neighbours, but there are still quite a few operations that might require quite a bit of power from the other peers, yet require no proof of work. A malicious peer could exploit these operations in a denial of service attack on the system. This could be easily prevented by requiring a proof of work for additional operations.

9 Further Work

Throughout this project, we have managed to meet our baseline goals for the system, and make a working command line chat client that is heavily secured. However, there is still much to be done before the system is enterprise-strength ready. Most importantly is a proper user interface. The current command line interface is not entirely intuitive, and lacks proper conversation threading.

Optimized k-walkers, such as Adaptive Probabilistic, could be a huge improvement. Currently our k-walkers are very basic, due to the lack of information about sender, receiver and content. Since we lack a lot of the information usually used to optimize k-walkers, it might prove difficult to make them perform better than they currently do, but it would be worth a try. Some of the available optimizations for k-walkers might also compromise the security of the system by introducing exploitable behaviour of the k-walkers.

A better cover traffic scheme could be implemented. The current cover traffic is completely random, and this leads to a slight possibility for traffic analysis against very active peers. Using constant-rate cover traffic would be preferable. In order to avoid eclipse attacks, it would be quite a good idea to avoid peers from the same subnet. Such a thing could be easily done by denying neighbour request from peers in an already connected subnet.

It would also be interesting to be able to send files using our system. But then if the file that was sent was very large we would need some mechanism to lower the amount of cover traffic to avoid putting to high a strain on the network.

Another functionality we could add could be group chat. This would make it easier for peers to share the same information with a group of other peers at the same time.

These improvements are only a few of many, there are countless of small changes that would make the system both safer, and more user friendly.

10 Conclusion

We managed to build a secure network for transmitting chat messages, using a fairly generic unstructured network as the base. The system is quite usable, and can send messages quickly enough to keep a conversation going, while still being secure against most attacks. We did this by following the design guidelines alluded to in the use-case, along with a much stricter plan specified in our internal design documents.

Not only does the system work, but we also managed to show experimentally that using proof-of-work for joining the network is a feasible method of preventing large-scale Sybil attacks. We have also shown how Diffie-Hellman encryption can be applied across a peer-to-peer network, with very few additional key negotiations. Finally, we have measured the performance of flooding versus k-walkers when very little information can be associated with the walkers, and have seen that as the network grows larger, k-walkers appear to still be competitive in terms of message count and latency.

We consider the project a success, since we reached our goals without any major hiccups, but we acknowledge that the system is still far from finished.

References

Ian Clarke, Scott G Miller, Theodore W Hong, Oskar Sandberg, and Brandon Wiley. Protecting free expression online with freenet. *Internet Computing, IEEE*, 6(1):40–49, 2002.

Jonathan Warren. Bitmessage: A peer-to-peer message authentication and delivery system. 2012.

Kris V. Ebbesen, Jan H. Knudsen, and Roland L. Pedersen. Peer to peer networks. 2014.

Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. Making gnutella-like p2p systems scalable. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '03, pages 407–418, New York, NY, USA, 2003. ACM. ISBN 1-58113-735-4. doi: 10.1145/863955.864000. URL `http://doi.acm.org/10.1145/863955.864000`.

Mihir Bellare and Phillip Rogaway. Optimal asymmetric encryption. In Alfredo Santis, editor, *Advances in Cryptology - EUROCRYPT 94*, volume 950 of *Lecture Notes in Computer Science*, pages 92–111. Springer Berlin Heidelberg, 1995. ISBN 978-3-540-60176-0. doi: 10.1007/BFb0053428. URL `http://dx.doi.org/10.1007/BFb0053428`.

Jonsson and Kaliski. Public-key cryptography standards (pkcs) #1: Rsa cryptography specifications version 2.1. 2003.

Whitfield Diffie and Martin E Hellman. New directions in cryptography. *Information Theory, IEEE Transactions on*, 22(6):644–654, 1976.

Michael J Freedman and Robert Morris. Tarzan: A peer-to-peer anonymizing network layer. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 193–206. ACM, 2002.

Adam Back. Hashcash - a denial of service counter-measure. 2002.

Python Software Foundation. Python xml vulnerabilitie. 2014.