Peer to Peer Networks

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

May 14, 2014

Contents

1 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.

2 Method of Operation

2.1 Base System

The system described is built atop the unstructured network developed during the P2PN course (TODO: ref earlier P2PN report). 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.

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.

2.2 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 (TODO:Mayby reference?) 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.

2.3 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** (TODO:Some sort of reference).

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, that only after obtaining the decrypted message will it be possible to verify the signature. This ensures 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. Also, should anyone attempt to tamper with the message before delivery, the signature will no longer be valid.

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. Also, should anyone tamper with the message before delivery, the receiver will no longer sign the correct data, making this easily detectable for the sender.

2.4 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.

2.5 Cover Traffic

Preventing traffic analysis by way of providing cover traffic is an important part of the network. 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.

2.6 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 (**?**), 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 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 likely, 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.

Note that the standard settings for the required bits of a proof of work are currently quite low, in order to allow rapid testing of the network

2.7 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.

2.8 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.

3 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 (TODO:Ref P2PN paper).

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 an 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 an RSA public key in the *pem* format.

publish ***public_key*** Make the given public 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 the have the corresponding hash string. Note that the key must be an 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.

References

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`.

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