A Cryptographically Secure Decentralised Communication Protocol

James Nicholson[1]

[1] University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge – CB3 0FD – United Kingdom

james.nicholson@cantab.net

Abstract. *This paper describes the implementation, as an instant messaging client, of a decentralised communication protocol, employing strong cryptography between nodes. This is achieved by building upon the OpenChord Distributed Hash Table (DHT) implementation, with a hybrid cryptosystem for mutual authentication and communication, using the Needham-Schroeder-Lowe public-key protocol, RSA, CBC-mode AES, and HMACs for integrity verification of messages.*

1. Background

Every day, millions of people communicate electronically via the Internet. The vast majority of this communication is transmitted in the clear[Mannan and van Oorschot 2006].Much of this communication relies upon centralised systems, requiring that users give the owners of these systems implicit trust if they are to communicate at all.

A centralised system means having a single point of control and a potential single point of failure in the network. As most Instant Messaging (IM) clients do not use end-to-end encryption, communication may be trivially intercepted at relaying servers by authorities and other eavesdroppers. This is a major concern when communicating important or secret information, or in locations where freedom of speech is restricted.

Some partial solutions do exist, for instance Skype offers end-to-end encryption for VoIP, however it does so in a closed way, with a central login server, leaving it vulnerable to DoS attacks. This was seen in 2007, when an update that was rolled out to all users at once caused every client to disconnect and reconnect, causing a DoS for 220 million users[Fildes 2007].

What is needed is a completely decentralised application, employing strong, end-to-end encryption, in order to allow secure communication where freedom of speech is restricted. The application must be resistant to censorship or compromise, either by repressive governments, or by malevolence (or foolishness) on the part of any controlling organisation. My intention has been to implement such a system.

This paper begins with a very brief background in peer-to-peer networking in section 2, followed by an explanation of the choices made for my cryptographic implementation, with the basic concepts and protocols and my reasoning for their use set out

in sections 3 and 4. A number of attacks are considered at each stage, with solutions to these laid out and their implementation discussed. Sections 5, 6 and 7 deal with the network aspects of the project, such as packet structure and the use of a distributed storage schema, whilst section 8 details the testing strategy for network code. Finally, sections 9 and 10 explore the limitations of my implementation, and present ideas for future work in developing this project.

2. Peer-to-peer Networks

One of the most common applications for decentralisation is in peer-to-peer networks, the majority of which are based on Distributed Hash Tables (DHTs). DHTs allow a hash table (a set of *(key, value)* pairs) to be distributed across a set of nodes (meaning clients in a system, in this example the computers connected in the network), allowing any node in the DHT to retrieve the value from a given key.

My implementation is based upon the Chord DHT[Stoica et al. 2001], in which nodes are arranged in a ring, and assigned a 160-bit identifier (a SHA-1 hash). In an n-node network, nodes keep a list of *O(log n)* other nodes, allowing lookups to be performed with *O(log n)* messages.

The Chord protocol automatically adjusts its internal tables to reflect newly joined nodes as well as node failures, which ensures that the node responsible for a key can always be found.

3. Identity Verification and Key Exchange

The Needham-Schroeder public-key protocol provides a simple means for two parties to each verify the identity of the other. I have chosen to use a modified version of this protocol, the Needham-Schroeder-Lowe (N-S-L) protocol, which includes a change that fixes a vulnerability to a man-in-the-middle attack[Lowe 1995].

1. $A \rightarrow B : \{N_A, A\}_{K_B}$
2. $B \rightarrow A : \{N_A, N_B, B\}_{K_A}$
3. $A \rightarrow B : \{N_B\}_{K_B}$

The N-S-L protocol has been proven correct [Backes and Pfitzmann 2004], and formally verified using a HOL (Higher Order Logic) theorem prover[Paulson 1998].

The protocol requires public and private keys for encryption and decryption, and therefore a public-key algorithm is required. As it is used as part of a hybrid cryptosystem, a symmetric-key protocol is also required, whose key I can exchange as a nonce in the above protocol. This requires that the symmetric-key protocol uses keys that are random, so as not to interfere with the behaviour of the key-exchange scheme.

RSA was chosen as my public-key protocol for use with N-S-L, and AES as the symmetric-key protocol, both implemented using the Java Cryptographic Extension (JCE).

N-S-L was chosen in order to make use of public-key cryptography to give each user a permanent identifier. This gives users the ability to prove their identities to one another, and offers the possibility in future to implement some form of PKI for users to create webs of trust.

In implementing the N-S-L protocol itself, I discovered that there was no standard library in Java, thus I have had to implemented it from scratch as two classes, for the two situations in which the application is required; either initiating the protocol as party A, or responding to it as party B. Each class contains generation and verification methods for each step.

The reason for the lack of such a library appears to be due to the wide adoption of Kerberos (which is available as a standard library), which seems to have been considered sufficient so that no N-S-L implementation is included.

I have therefore ended up spending a great deal of time implementing the protocol myself and testing it heavily to ensure correct behaviour. Whilst this has meant more of my time was spent focusing on the core of the application (leaving less time to persue the ideas presented in sections 9 and 10), it has allowed me to become much better acquainted with the various attacks on protocols, and the processes that led to the development of NS-L.

3.1. Nonces and Identifiers

N-S-L requires nonces to prevent replay attacks, etc. To generate these I have used Javas `SecureRandom` class, which was chosen because it provides a cryptographically strong pseudo-random number generator, complying with statistical tests set out in FIPS 140-2 (section 4.9.1) and RFC 1750[Eastlake, 3rd et al. 1994].

A 256-bit nonce size was chosen because of its use as part of a hybrid cryptosystem: nonce N_A is replaced with two 128-bit blocks, used as the shared symmetric (AES) key and message authentication (HMAC) key (discussed in section 4.1) respectively. Using the keys in this way does not open new side channel attacks, as both the symmetric and authentication keys can be any random series of bytes, with no constraints placed on their selection.

N-S-L requires both parties in the protocol to have an identifier (which could, for instance, be the users RSA public key). As Javas `Cipher` class does not permit encrypting multiple blocks using RSA (and splitting up these messages could be insecure), each N-S-L message is limited to the length of the RSA key[1]. Therefore, using each partys RSA public keys as an identifier is not possible, as these are longer than than the maximum block size.

A solution to this is for each party to use a fingerprint (in this case a hash of their public key) as an identifier, which can be verified by the receiver using the same hash function. This requires a preimage resistant hash function, and for this I have chosen the SHA-256 algorithm as it provides the largest output hash size that will still fit inside the key-exchange protocol. This is implemented using Javas `MessageDigest` class.

4. Communication Encryption

Naturally, public-key cryptography would be too slow for instant messaging itself. Instead, I have selected the Advanced Encryption Standard (AES) for symmetric-key encryption due to its wide use and adoption as a standard by the National Institute of Standards and Technology[National Institute of Standards and Technology (NIST) 2001]. To

[1] Minus the bits required by the padding scheme.

prevent the possibility of certain types of replay attack, I have chosen the Cipher-Block Chaining (CBC) mode of operation.

CBC mode requires an Initialisation Vector (IV). It is not necessary to encrypt the IV, so this can be prepended to each message to be used be the receiver. To minimise the possibility of IV collisions, which could allow an eavesdropper to detect that the same message had been sent twice, the IV is 128 bits in length. I do not worry too much with the IV, since it could even be fixed, since we user randomly generated key derivated from the nonces. My AES implementation makes use of the JCE, and provides methods to generate AES keys and initialisation vectors, used by my cryptography API when generating messages for communication.

4.1. Message Authentication Code

To allow the integrity of messages to be verified, each message includes a Message Authentication Code (MAC) to allow the remote user to verify that the content of a message has not been modified in transit. When combined with a cryptographic hash function and secret key, these are called keyed-Hash Message Authentication Codes (HMACs). HMACS are defined in RFC 2104[Bellare et al. 1997], and described below.

$$HMAC_{Km}(m) = h\Big((K \oplus opad), h((K \oplus ipad), m)\Big)$$

Where h is a cryptographically secure hash function, K_m is a secret key, $ipad$ and $opad$ are fixed strings used for padding, and m is the message to be authenticated.

As with AES, the secret key may be any random series of bytes, and can thus also be exchanged during the handshake. The HMAC is applied to the block consisting of the encrypted message, identifier and counter (described in section 4.2) to prevent replay attacks.

$$HMAC_{Km}\Big(\{m, i, c\}_{K_{AB}}\Big)$$

Javas own HMAC library does not make it possible to cast between HMAC keys and byte arrays, as it does not include methods to construct keys from a given input, unlike most other methods. As I require the ability to convert to and from streams of bytes for transmission over a network, I implemented the above HMAC definition directly using RFC 2104[Bellare et al. 1997], using byte arrays as keys, and verified my implementation by testing it against NISTs reference figures.

4.2. Preventing Attacks

By itself, an HMAC will not prevent replay attacks. Therefore an attacker can replay a message which would be accepted by the receiver as valid, breaking my message integrity requirements. In order to prevent this, a counter is added to the message before encryption, and incremented each time a message is sent. Any message whose counter value is not strictly greater than the previous message is rejected. Any message whose counter value is not as expected will cause the application to issue a warning that a message has been missed.

I have used the Java `BigInteger` class to provide a counter, as it allows arbitrarily large numbers that do not wrap around, and simplicity of conversion to and from

byte arrays. The counter is only reset when a new session is started (that is, when both the symmetric and HMAC keys are changed).

In addition to a counter, the encrypted message also requires an identifier for the sender. This is to prevent a replay attack, possible by a race condition, in which a sender might have his own message replayed to him, as shown in the example below.

1. $A \rightarrow B : \{m, c\}_{K_{AB}}, h$
2. $C \rightarrow A : \{m, c\}_{K_{AB}}, h$

Where m is the message, c is the counter and h is the HMAC. In the above example, C eavesdrops on As message to B, and sends it back to A. As the message has a valid HMAC and correct counter value, A believes that the message is from B. By adding an identifier, A can see that the message is his own, and will reject it; this is the property we expect when using HMACs for verifying authenticity[Anderson 2001].

5. Packet Structure

As packets are to be transmitted over a network, they must to have a well-defined structure, in order that they can be correctly constructed and read by both parties. The structure of the key exchange and communication messages is described in sections 5.1 and 5.2 respectively. Figure 1 gives an overview of the communication message structure.

5.1. Handshake Packets

The N-S-L handshake packets consist of a combination of nonces and identifiers, represented by N_X and X respectively, where $X$is the relevant party. The block is then encrypted using the other partys public key,K_X.

1. $A \rightarrow B : \{N_A, A\}K_B$
2. $B \rightarrow A : \{N_A, N_B, B\}K_A$
3. $A \rightarrow B : \{N_B\}K_B$

Both nonces and identifiers are 256 bits in length, thus the longest message (step (2) above) is 768 bits long, below the maximum 1024-bit message length (minus the padding scheme) for my RSA implementation.

5.2. Communication Message Packets

The `PacketStructure` class provides the implementation for the structure of message packets, and contains methods both for encoding and decoding these packets.

$$A \rightarrow B : IV, \{Identifier, Counter, Message\}_{K_{AB}}, HMAC$$

Messages consist of an Initialisation Vector (IV), an encrypted block consisting of the identifier, a counter, and the message encrypted with the shared key K_{AB}, and the Message Authentication Code (HMAC) of the encrypted block (Figure 1).

6. Distributed Storage Schema

In order to be searchable and accessible, each user must store a set of values in the DHT to allow other users to retrieve their details and initiate connections. Each user is identified

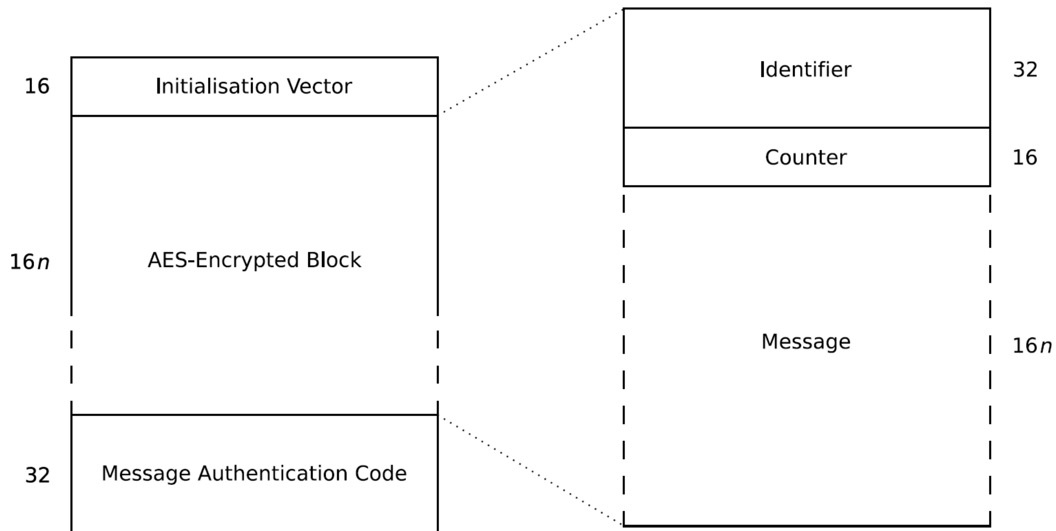

Figure 1. Message packet structure.

by their key fingerprint (a 256-bit hash of the users public key), which can be used to retrieve the users public key, IP address, DHT port and connection port.

$$Identifier \rightarrow PublicKey, IPAddress, DHTPort, HandshakePort$$

Performing a DHT lookup on an identifier will return a `HashMap` object containing the above values. This detail is hidden within the API, which writes all values when the DHT object is constructed, and provides methods to the user to retrieve each of the values using the identifier.

These methods include validation of the values returned, for instance checking that the returned public key does indeed hash back to the identifier used to perform the lookup, to prevent attacks based on modification of the DHT. More attacks in this vein are discussed in section 9.2.

7. Connection Handling

Once a node has joined the DHT, the `ListenerThread` class is instantiated. This waits for incoming connections and starts an instance of the `Handler` class each time a connection from another node is made. The handler thread then performs the handshake and communication using the cryptography API.

When initiating connections, the module uses the DHT API to perform a lookup of the address of the remote node, and again hands over to the cryptography API to initialise the handshake and allow communication.

As noted above, the `ListenerThread` class is responsible for controlling instances of the `Handler` class. Each handler is responsible for a different connection, allowing communication with many users at once. The handler is responsible for performing its part of the handshake (via the cryptography API), and runs until one of the users ends the communication by closing that part of the user interface.

7.1. IP Address Issues

Java has a known issue with GNU/Linux systems[2] in which Javas method for determining a computers external IP address is ambiguous, and often returns the loopback address, which is not helpful when reporting ones IP address to other nodes.

As a partial solution, I have implemented a method to enumerate all addresses of all network cards connected to the computer, which the user interface then allows the user to choose from (or provide a different IP address if necessary).

8. Testing with PlanetLab

PlanetLab is a geographically distributed overlay network designed to support the deployment and evaluation of planetary-scale network services.[Bavier et al. 2004]

Stability and resilience testing of network and decentralisation code included use of PlanetLab, deploying my code across hundreds of nodes around the world. In order to make use of the PlanetLab service, a user requires a slice, a virtual operating system spread across the many hundreds of nodes in the PlanetLab network. This was very kindly provided by Dr. Steven Hand.

9. Limitations

I am aware of a number of shortcomings of my project that I have been unable to address properly due to time constraints, and these are discussed here.

9.1. Formal Proof of Correctness

Whilst the individual parts of my cryptographic implementation have been carefully tested for correctness, I have not shown that this is true for the composition of these elements. A formal proof of correctness for the symmetric-key portion of my communication protocol would allow me to be certain that no attacks against it are possible[Wang et al. 2006]. However, after discussion with my supervisor it appears that doing so would at best take several months to complete and require learning far beyond the scope of the undergraduate course.

9.2. Denial of Service in DHT

As each node is responsible for a certain proportion of entries in the DHT, and as any user is able to modify an entry whose key he knows, a malicious user might create a Denial of Service (DoS) in cases where he is in control of an entry requested by another user. While this does not compromise the security of communication (public keys are checked against identifiers by way of a cryptographically secure hash function), it might prevent communication in the first place if a user is unable to retrieve another users details, or is given incorrect details which are then rejected.

This could be partially solved by key-replication (a side effect of increasing network stability, discussed above), however this still does not guarantee that DoS is not possible, it merely makes it less likely: as the number of replications is increased, the probability that all replicas are controlled by malicious nodes drops off sharply. This also requires removing the contacts list functionality, whereby users can see all connected nodes in the network (as opposed to just user-input contacts), to prevent a malicious user iterating through the list and modifying DHT entries and replicas.

[2] Bug #4665037, InetAddress.getLocalHost() ambiguous on Linux systems

9.3. Denial of Service in Communication

Decryption of received messages can lead to a denial of service if a malicious user sends data more quickly than the application can decrypt it. This can be mostly solved by modifying the protocol in such a way that the sender must do more work than the receiver for communication. This might involve the sender calculating certain checks on the message which are more resource-intensive to generate than to verify by the receiver.

9.4. Public Keys

My application is based around a user-centric trust model, where users are responsible for exchanging public keys in a trusted way if they are to trust one anothers identities when communicating.

As my requirement is for a completely decentralised system, I was unwilling to provide public key servers, since these require either relying on a centralised system, or implementation in such a way that keys can be signed in a decentralised system. This would mean a great deal of research into attacks on such systems (as well as DoS issues), and implementation of a very large piece of PKI functionality that was not a part of my main goal.

Vulnerabilities to consider in such systems include Sybil attacks, based on subverting reputation systems[Douceur 2002], and thus range far beyond cryptographic attacks.

9.5. Plausible Deniability

In the course of this project, I researched quite seriously into implementing functionality for plausible deniability, as this might be particularly valuable to users operating where freedom of speech is restricted.

Depending on how it is implemented, plausible deniability can mean several things for the user, such as giving users the ability to plausibly deny that they sent or received a message, that it originated with them rather than simply being routed through them, or that they can decrypt a given message.

One possibility for providing this is to implement anonymous routing between nodes, such that it is not possible for an eavesdropper to know who is communicating with whom. Sadly, after conversation with Stephen Murdoch (a member of the Tor project[3]) it seems that implementing anonymous routing is significantly beyond the scope of an undergraduate project.

A simpler means of providing some small amount of deniability might have been implemented from the outset by using Diffie-Hellman key exchange for session keys, which would mean that if a users long-term private key is compromised it still cannot be used to decrypt the communication between users[Diffie and Hellman 1976]. Another option was MQV (Menezes-Qu-Vanstone)[Menezes et al. 1995], an authenticated protocol for key agreement based on the Diffie-Hellman scheme, substituting then the whole NS-L scheme.

[3]Tor provides anonymity by routing traffic via a series of nodes in the Tor network, using incremental encryption such that no single node knows a packets complete route. See http://www.torproject.org/

As it stands, communication can be reconstructed at a later date by an eavesdropper if either users private key is compromised, and I feel that this is a flaw in my implementation.

10. Future Work

This project offers a number of extremely interesting avenues for extension, some of which I have researched in-depth, while others remain unexplored. The more immediate extensions that I would wish to implement given time are listed below.

- **Diffie-Hellman** As discussed above, implementing Diffie-Hellman key exchange would provide a simple and effective form of plausible deniability for information communicated where long-term private keys are later recovered. Also we can study to change the whole sheme for a MQV one.
- **Tor** Routing my application over the Tor network would provide a much greater level of plausible deniability, as users can deny ever sending messages to one another. This would require overcoming several hurdles, not least the operation of the DHT.
- **Resilience** Replication of entries across multiple nodes greatly increases the resilience of the network as a whole, as multiple nodes are responsible for keys and restabilising becomes less important. This could be implemented without significant degradation of performance by performing lookups in parallel.
- **Preventing Denial of Service** Implementing replication of entries also provides some protection from DoS vulnerabilities in the DHT, as incorrect entries can be identified by users and rejected. Denial of service in communication can be dealt with as described above, by modifying the protocol to make the sender do more work than the receiver.

In the long term, expanding my application APIs to provide a base on which other applications can be built would allow for a much wider variety of uses, such as providing a distributed data store, peer-to-peer functionality, or multimedia communication, such as voice chat.

References

[Anderson 2001] Anderson, R. J. (2001). *Security Engineering — A Guide to Building Dependable Distributed Systems*. John Wiley & Sons.

[Backes and Pfitzmann 2004] Backes, M. and Pfitzmann, B. (2004). A cryptographically sound security proof of the needham-schroeder-lowe public-key protocol. *IEEE Journal on Selected Areas in Communications*, 22(10):2075–2086.

[Bavier et al. 2004] Bavier, A. C., Bowman, M., Chun, B. N., Culler, D. E., Karlin, S., Muir, S., Peterson, L. L., Roscoe, T., Spalink, T., and Wawrzoniak, M. (2004). Operating systems support for planetary-scale network services. In *NSDI*, pages 253–266. USENIX.

[Bellare et al. 1997] Bellare, M., Canetti, R., and Krawczyk, H. (1997). HMAC: Keyed-hashing for message authentication. Internet Request for Comment RFC 2104, Internet Engineering Task Force.

[Diffie and Hellman 1976] Diffie, W. and Hellman, M. E. (1976). New directions in cryptography. *IEEE Transactions on Information Theory*, 22(5):644–654.

[Douceur 2002] Douceur, J. R. (2002). The sybil attack. In Druschel, P., Kaashoek, M. F., and Rowstron, A. I. T., editors, *IPTPS*, volume 2429 of *Lecture Notes in Computer Science*, pages 251–260. Springer.

[Eastlake, 3rd et al. 1994] Eastlake, 3rd, D., Crocker, S., and Schiller, J. (1994). RFC 1750: Randomness recommendations for security. Status: INFORMATIONAL.

[Fildes 2007] Fildes, N. (2007). Skype software glitch hits 220 million users. *The Independent*, (18 August 2007).

[Lowe 1995] Lowe, G. (1995). An attack on the Needham-Schroeder public-key authentification protocol. *Information Processing Letters*, 56(3):131–133.

[Mannan and van Oorschot 2006] Mannan, M. and van Oorschot, P. C. (2006). A protocol for secure public instant messaging. In Crescenzo, G. D. and Rubin, A. D., editors, *Financial Cryptography*, volume 4107 of *Lecture Notes in Computer Science*, pages 20–35. Springer.

[Menezes et al. 1995] Menezes, A. J., Qu, M., and Vanstone, S. A. (1995). Some new key agreement protocols providing mutual implicit authentication. In *Second Annual Workshop on Selected Areas in Cryptography (SAC '95)*, Lecture Notes in Computer Science, pages 22–32. Springer-Verlag, Berlin Germany.

[National Institute of Standards and Technology (NIST) 2001] National Institute of Standards and Technology (NIST) (2001). Advanced Encryption Standard (AES). Federal Information Processing Standards Publication (FIPS PUB) 197.

[Paulson 1998] Paulson, L. C. (1998). The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1-2):85–128.

[Stoica et al. 2001] Stoica, I., Morris, R., Karger, D., Kaashoek, F., and Balakrishnan, H. (2001). Chord: A scalable Peer-To-Peer lookup service for internet applications. In Guerin, R., editor, *Proceedings of the ACM SIGCOMM 2001 Conference (SIGCOMM-01)*, volume 31, 4 of *Computer Communication Review*, pages 149–160, New York. ACM Press.

[Wang et al. 2006] Wang, A., He, F., Gu, M., and Song, X. (2006). Verifying java programs by theorem prover HOL. In *COMPSAC*, pages 139–142. IEEE Computer Society.