Secure messaging within a decentralised anonymity network

Graham R. Armstrong
11004764

Software Engineering Project

A Report submitted in partial fulfilment of the
regulations governing the award of the Degree of
BSc (Honours) Computer Science
at the University of Northumbria at Newcastle

April 2015

Northumbria University Newcastle
Faculty of Engineering and Environment

ABSTRACT

In the modern information age, we have more devices connected to the internet than at any previous time—and more devices collecting our data. The right to privacy is a difficult right to maintain.

This dissertation focuses on the subjects of peer-to-peer networking and anonymity networks; specifically on the developing of an anonymity network specifically for instant messaging. A product called DistrIM has been produced and demonstrated over LAN.

Techniques for ensuring the security and the anonymity of users are researched and analysed. This project also looks at structured peer-to-peer networks, doing a comparison of distributed hash tables or DHTs, and implementing a basic DHT algorithm for the demonstration of the network.

Techniques for creating DHT-based networks are also analysed.

A relaying feature, inspired by Tor (The Onion Router), is implemented to allow users to send messages securely and anonymously.

The project successfully produces an application capable of forming networks and using those networks to implement an onion routing algorithm. The application is demonstrated by sending messages across a network via random nodes.

DECLARATION

I declare the following:

1. that the material contained in this dissertation is the end result of my own work and that due acknowledgement has been given in the bibliography and references to **ALL** sources be they printed, electronic or personal.

2. the Word Count of this Dissertation is **13,419**.

3. that unless this dissertation has been confirmed as confidential, I agree to an entire electronic copy or sections of the dissertation to being placed on the eLearning Portal (Blackboard), if deemed appropriate, to allow future students the opportunity to see examples of past dissertations. I understand that if displayed on eLearning Portal it would be made available for no longer than five years and that students would be able to print off copies or download.

4. I agree to my dissertation being submitted to a plagiarism detection service, where it will be stored in a database and compared against work submitted from this or any other School or from other institutions using the service.

   In the event of the service detecting a high degree of similarity between content within the service this will be reported back to my supervisor and second marker, who may decide to undertake further investigation that may ultimately lead to disciplinary actions, should instances of plagiarism be detected.

5. I have read the Northumbria University/Engineering and Environment Policy Statement on Ethics in Research and Consultancy and I confirm that ethical issues have been considered, evaluated and appropriately addressed in this research.

Signed:

ACKNOWLEDGEMENTS

I am immensely grateful to my project tutor David Kendall who has been instrumental throughout the course of this project. Thank you very much for your help and guidance David.

CONTENTS

ONE

INTRODUCTION

1.1 Aims and Objectives

The Aims and the Objectives, originally defined in the Terms of Reference (Appendix A), are the yardsticks by which the project and product are measured. The aims are the high-level goals, the objectives are the lower level goals that should be completed to fulfil the aims. A description and justification for the aims and objectives adopted for this project.

Aims

1. **To investigate how peer-to-peer anonymity networks protect the information shared by its users.**

   The focus of this dissertation project is on secured messaging within a decentralised anonymity network. The first aim was to complete the research necessary to make that possible.

   It was known that one of the requirements from the Proposed Work section of the Terms of Reference was to build the system as a peer-to-peer network which incorporates a Distributed Hash Table. The aim also makes a specification that the information shared by the product should remain secure.

2. **To build a prototype instant-messaging application in an anonymity network, allowing users to securely share messages.**

   This is the aim reached by the synthesis section and implementation. After the finalisation of the requirements specification, an appropriate initial design was created from which to begin development.

Objectives

- **Research implementations of peer-to-peer networks.**

  Types of peer-to-peer network were researched, specifically to look at different types of structured peer to peer networks.

- **Research techniques for maintaining anonymity.**

This involves research into anonymity and relaying techniques within peer-to-peer networks; specifically looking at TOR, The Onion Network and I2P The Invisible Internet Project.

- **Analyse DHT algorithms and to determine an appropriate one for the network.**

  Once the research into P2P DHT algorithms is completed, an algorithm should be picked out from the DHTs identified to base the overlay network on.

- **Create design documentation, defining the protocol for sharing messages.**

  UML diagrams show the design of the system. A class diagram showing the components of the system, and sequence diagrams showing the protocol in action.

- **Implement a DHT algorithm to construct a peer-to-peer network.**

  The DHT algorithm controls how nodes become aware of each other and connect to each other, however this objective was not met. A simple DHT has been created to connect all nodes to each other.

- **Implement a relay protocol to secure connections between nodes.**

  A relaying method will be identified to allow nodes to securely and anonymously send messages to one another.

- **Implement anonymity features.**

  Some anonymity features identified in the analysis should make their way through design and into the finished product.

- **Successfully connect nodes together in a network.**

  For testing purposes, it should be possible for nodes to connect to one-another and join each others networks.

- **Demonstrate a working network by showing nodes connecting to one another and transferring messages.**

  For testing purposes, it should be possible to connect nodes together into a network and then share messages between each other using the relaying methodology discussed above.

- **Evaluate the success of the network.**

  Tests have been designed to demonstrate the product functioning correctly.

1.2 The Product: DistrIM

The product that has been developed, named DistrIM, is a command line program that allows users to create networks of nodes that can chat securely with one another. Its purpose is to enable secure messaging between users by using anonymity methods to hide messages from the outside world.

DistrIM provides a command line interface, a message prompt appears to receive user input. Output will be displayed on the prompt too. The program allows users to view information about other nodes on the network.

A user with an active instance can send a message to a foreign node. By using the SEND command, with the identifier of the foreign node, a message will be sent securely using the novel onion routing algorithm.

When a user sends a message, it will be relayed via a random route of other nodes.

The recipient will receive the message and it will be displayed on the screen with a message to inform them of who its been sent from.

A list of arguments are available if the program is started with a **-h** option.

1.3 Context of the Problem

The Terms of Reference, section A.2, give a good explanation for the background of this project.

With the ubiquity of internet-enabled devices in the modern information age, there are more devices dealing with our data than ever before. The problem this presents is that security cannot be guaranteed. What is of particular concern are the revelations of mass snooping that have surfaced as a result of the leaks by whistle-blower Edward Snowden. We know that corrupt governments are prone to enacting censorship or cracking down on supposed dissidents. An example of this is the Arab Spring, when the Egyptian government blocked access to Facebook and Twitter in an attempt to quell protesters.

The project will focus on the technology of anonymity networks. Anonymity networks are peer-to-peer networks that use novel relaying methods to mask the source and destination of any data it handles.

Peer-to-peer networking is a method of sharing resources directly between devices, called nodes, without the need for centralised servers. This differs from many internet services which are often setup as the default client-server model.

A real life example of an anonymity network is Tor (The Onion Router). Tor helps people access web content online they might otherwise not have access to due to censorship.

This project seeks to apply these ideas to an instant messaging application, allowing users to send messages to one another that are routed via other nodes.

Peer-to-peer networks can be classified into one of two types of overlay network, structured or unstructured. Unstructured networks take their shape from the nodes that join and leave. This project will focus on structured networks in particular, those that make use of Distributed Hash Tables.

1.4 Approach Summary

This project was developed in Python, making use of its object-orientated functions. The system will be designed in UML.

TWO

ANALYSIS: LITERATURE REVIEW

2.1 Distributed Hash Table

A Distributed Hash Table, or DHT, is a method of sharing data between nodes in a structured peer-to-peer network. Data within the table is stored in key, value pairs such that the key maps to the value (key $\rightarrow$ value). A DHT can be used for many varieties of distributed products and services, the values within the table can represent multiple types of information such as files or nodes.

2.1.1 DHT Protocols & Algorithms

The topology of a structured peer-to-peer network is such that strict precise rules govern how lookups for nodes are performed. As such, it should always be possible to locate a key in the network; regardless of faults and within a reasonable length of time. (Cheng-Zhong, 2005)

Using DHT for the set-up of a service yields several potential benefits. One of those benefits is redundancy, if a peer-to-peer network loses nodes (either through failure or through those nodes simply disconnecting) then it should be possible for nodes to route around those failures and continuing functioning.

In order to understand how DHT operates, it's first important to define terms common to DHT algorithms:

Node
: A node is a device inside the peer-to-peer network; in typical examples it is represented by a desktop Personal Computer however any internet-enabled device can form part of a P2P network, including mobile phones and tablets.

Node ID
: A node ID is an identifier which is unique to a specific node on a network. The node ID is used by nodes in order to locate and communicate with one another. In DHT-based P2P networks, this ID informs other nodes where the node is in the overlay network.

Overlay Network
: The overlay network is an abstract virtual network which is built upon an existing physical computer network, typically the internet. The overlay network represents the way the DHT is structured, thus influencing the routes necessary for nodes to find one-another.

Lookup

A lookup is an action performed by a node in an attempt to match a key in the DHT to a value. This takes the form of a function for which a key is passed in and the value is returned.

Hashing

Hashing is the method by which the key of a value is calculated.

Finger

A finger is a data structure used by DHT implementations to store information about foreign nodes, typically the IP address which is mapped to by the Node ID. Any additional node-specific information can be stored here too.

Finger Table

The Finger Table is a data structure used by DHT implementations to store collections of Fingers. It may be referred to by other names, such as the routing table.

For a node to make use of a P2P network it must first find other nodes on the network. Once connected the node can use a process called discovery to ask other nodes which parts of the resource or service they have; if a resource is needed then a node can perform a lookup, which is the process for which a node asks other nodes if they have a particular resource, and if that resource is found on another node, the two nodes can establish a connection with one another to begin the data transfer.

2.1.2 DHT Algorithm: *Chord*

One of the de-facto standards of DHT algorithms is Chord, it is one of the earliest examples of a DHT algorithm, having been first written about in 2001. The single operation of Chord is the lookup, to map a key to a node. (Stoica et al., 2001)

The structure of a Chord-based overlay network could be fairly described as a ring. The position of each node in the ring is dependent upon its node ID. Each node has a direct successor, that is the node with the next ID in the series; the direct successor of the node with the highest ID is the node with the lowest ID, hence forming a ring. Unlike traditional name and location services such as DNS, the physical location is not important to the network structure.

Each node in a Chord network has a single, unique ID or key, which other nodes in the network will use to locate and identify it. These keys are determined by a process of *consistent hashing*. Keeping the network balanced in this way can help avoid the problems of overloading nodes with routing data and having to perform the costly operation of rebalancing. Hashing is discussed in more detail in subsubsection 2.1.5.

The overlay network structure shown in Figure 2.1 means nodes can *hop* between other nodes to contact unknown nodes without the need to know about every node in the network. This technique drastically improves scalability, thus meaning that it is possible for networks to become very big while remaining functional. The Finger Table of a Chord node need only be the size of $logN$, and routing messages is a $O(logN)$ operation, where N is the number of nodes in the network.

When seeking a node in a Chord overlay network, the node performing the lookup request will

Chord Routing Operation	
Node ID	Finger Table IDs
012	013, 031, 098, 106, 202, **420**, 781
420	421, 429, 453, **560**, 652, 699, 880
560	561, 572, **591**, 643, 789, 902, 002
591	592, **642**, 643, 756, 989, 002, 012

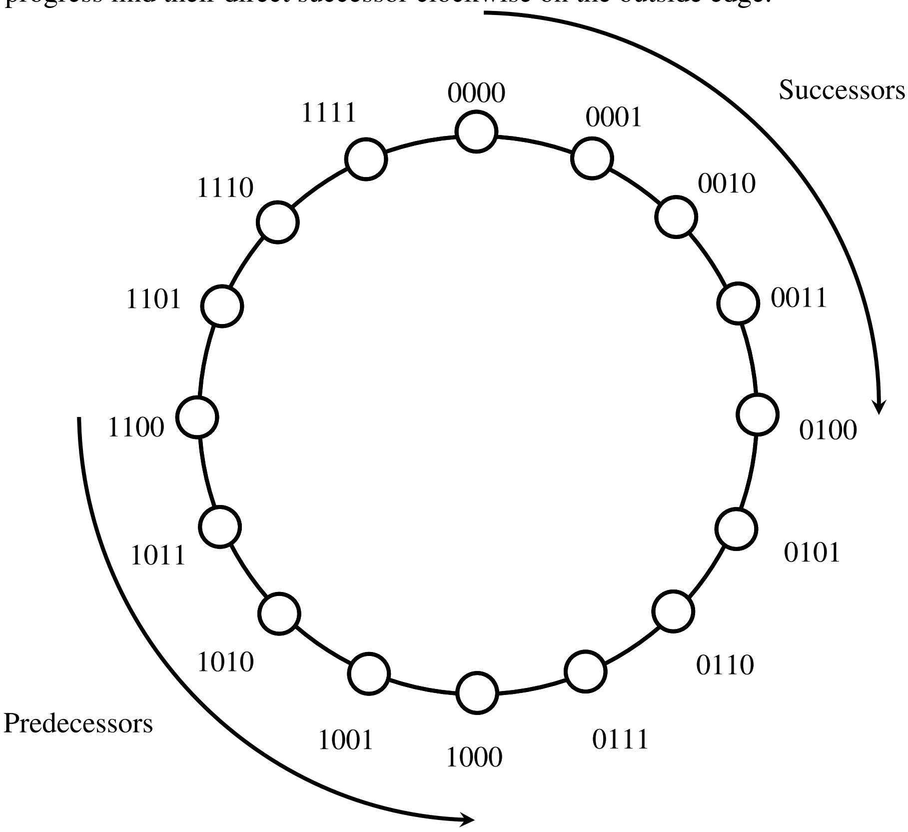
Tab. 2.1: An example of a Chord lookup operation, demonstrating the hops needed to get from node 012 to 642.

attempt to seek out a node behind the target node. Table 2.1 shows an example of a chord routing operation between 4 nodes, the bold labels show the node ID of the next chosen hop.

2.1.3 DHT Algorithm: *Pastry*

Pastry is another common DHT algorithm. (Rowstron and Druschel, 2001)

Fig. 2.1: A ring-shaped overlay network, the basis for Chord and Pastry. Successive nodes progress find their direct successor clockwise on the outside edge.



Like Chord, the overlay network is ring-shaped, nodes have direct successors which are used to perform node lookups. Unlike Chord however, Pastry provides bi-directional routing in the overlay network, as nodes are aware of their predecessors as well as their successors.

The Finger Table of a Pastry node is composed of three parts: the leaf set, the neighbourhood set, and the routing table. The leaf set stores the finger of nodes whose identifiers are directly before or directly after the identifier of the current node. The neighbourhood set contains the finger of nodes that are closest to the current node based on the *proximity metric*. The routing table stores information about nodes that are not nearby in the overlay network or nearby in terms of the proximity metric, but can be used for *routing* messages

One of the interesting aspects of Pastry, compared against Chord, is that it provides network locality using a proximity metric. The *proximity metric* used depends upon the implementation, but typically it is defined as the number of routing hops needed to get from the local node to the foreign node. This feature is valuable for distributed services that deal with large amounts of data transfer, such as file sharing, since reducing the distance and latency between devices often shortens the time necessary to share data. This is most certainly not ideal from an anonymity point of view however, as any predictability provided by the network protocol could be used to determine private information about nodes in the network.

Pastry provides a routing algorithm for sending messages to nodes on the network in the lowest number of hops possible. The best case scenario is trying to contact a foreign node that falls within the local node's *leaf set*, since if a node is part of this leaf set then the finger of that node will already be available in the local finger table. Otherwise the routing table must be used.

The closest node to the target is the one that shares the longest common prefix. For instance if the target node for a lookup has a node ID of 01010011_2 and the routing table has three nodes: 11010011_2, **01**110000_2, and **0101**1101_2, the request will be routed through the final node because it has the longest common prefix of four bits (highlighted in bold). Like Chord 2.1.2 this lookup is an $O(logN)$ process.

The routing scheme of Pastry is described as deterministic, and as such it is acknowledged by the authors that it is vulnerable to failed or even malicious nodes that may be on a given routing path. This is discussed in subsubsection 2.1.5.

2.1.4 DHT Algorithm: *Kademlia*

Kademlia is another DHT algorithm that differs quite significantly from Chord and Pastry. It offers a $O(logN)$ lookup operation like Pastry but using a larger finger table. (Maymounkov and Mazieres, 2002)

The Kademlia overlay network is structured as a binary-tree rather than a ring structure. The tree can be thought of as a search path based on the bits of the Node ID, traversing left on a zero and traversing right on a one until reaching the bottom of the tree; the path of edges taken to the bottom represents the Node ID, and thus the bottom layer of the tree represents the nodes in the DHT network.

Fig. 2.2: A tree diagram representing the overlay network of a basic Kademlia network with a 3-bit identifier.

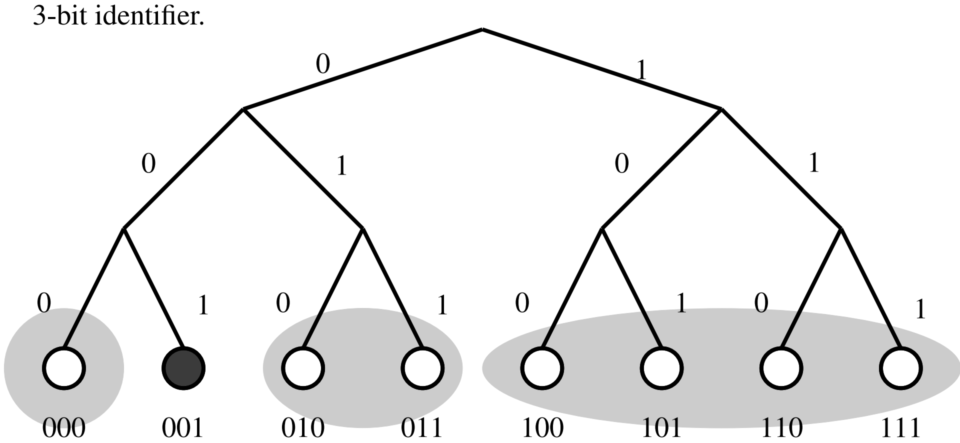


Figure 2.2 shows how the XOR metric splits the network into partitions. The sections highlighted in grey represent the partitions according to node 001; on the far left is the node with the longest common prefix, sharing a common 00. Node 001 would need to be aware of either node 010 or 011 to make that section of the tree visible. Node 001 would only need to be aware of one node in the far right tree, as any node with no common prefix should be able to route into the partitions it has made for itself on its side of the tree.

The *routing table* in Kademlia is made up of *k-buckets*, which are lists of node fingers. The collections of lists themselves are arranged in a tree structure, each list will contain a list of nodes with a common prefix; the list on the far left of the tree will contain all nodes beginning with the common prefix 000, whereas the list on the far right of the tree will contain all nodes with the common prefix 111. These lists are assigned dynamically, so if there are no nodes in the network with a given common prefix, no k-bucket will have been assigned to accommodate them. To help maintain even an unbalanced network, full k-buckets can be split into k-buckets with larger common prefixes.

The k-bucket system helps to provide greater level of resistance against routing attacks than Chord or Pastry, due partly to redundant links. Kademlia makes use of additional finger information which is passed around with messages. When a new node is introduced, it is added to the appropriate k-bucket; if however the finger is already present in the k-bucket then its finger is moved to the top of the list. Having redundancy built into the system this way means any node can easily and quickly use an alternative path to relay a message should a node disconnect, graciously or otherwise. Note that nodes are only removed from a k-bucket if they leave the network, and only if they are at the bottom of the list; attempts to flood a node with fingers won't prevent that node using its relay information. (Urdaneta et al., 2011)

Kademlia implements a common-prefix oriented routing algorithm in much the same way as Pastry 2.1.3 does; however the unique tree-based XOR-topology means that complexity is reduced, and the need for secondary neighbourhood sets and leaf sets like in Pastry is removed. While routes devised from this algorithm are uni-directional like in Chord 2.1.2, the distance in the network between two nodes is symmetric; or rather there are two different paths from Node A to Node B, and Node B to Node A respectively, but both those paths hop across the same number of nodes.

2.1.5 Facets of DHT Algorithms

Hashing

As previously identified, DHT algorithms provide a way of mapping node information using hash values. An important consideration for the design of DHT protocols is which algorithm to use for completing the hash.

The basic premise of a hashing algorithm is that any data, of any length, can be reduced to a set-length bit-pattern, called a *hash value* or *digest*, that represents the data. Hashing functions are not random, for a given hashing function the input $data_a$ will always give the output $digest_a$. Hashing algorithms tend to compute results in rounds, so even small changes in the data such as flipping a single bit are likely to produce vastly different digests.

There are an infinite number of data combinations, but a finite number of digests, so it is possible for two distinct data values to correspond to the same hash value; this is called a collision. A good hashing function will avoid collisions as much as possible.

One of the most common hashing algorithms used in computing is MD5. It's most common usage is in file verification, ensuring that a downloaded file keeps its integrity after the download. MD5 is an extension of the MD4 hashing algorithm that gives a greater focus to cryptographic integrity and collision avoidance. (Rivest, 1992) MD5 has been used for password storage in the past, however its use is no longer recommended. MD5 has been found to be cryptographically insecure, an attack was discovered by Xie et al. which allows an attacker to easily find a collision. For instance a valid packet of data can be substituted for an invalid or malicious packet of data, and be engineered to produce the same hash value as the valid packet; hence successfully tricking the verifier into believing the data is legitimate. (Xie et al., 2013) This could be a problem for DHT networks, as an attacker could exploit the hashing algorithm to allow them to imitate another node.

Karger et al. addresses the needs of hashing algorithms in distributed systems and proposes caching protocols to alleviate the problems suffered by rudimentary DHT algorithms. *Consistent Hashing* is a technique intended to vary the locality of nodes within a DHT network by ensuring hashed values are randomly spread across the range of values. This is important to avoid *hot-spots*, nodes that share a greater burden than others for dealing with queries in the network. (Karger et al., 1997)

The Chord DHT protocol uses consistent hashing with the SHA-1 hashing function as its base. (Stoica et al., 2001) Pastry and Kademlia also use SHA-1. (Rowstron and Druschel, 2001) (Maymounkov and Mazieres, 2002) SHA-1 is considered more secure than MD5, and more difficult to cryptographically break, but attacks on SHA-1 do exist. Because of this, it is recommended to use the newer SHA-2 algorithm for security purposes. (Wang et al., 2005)

Churn

One issue that can blight large-scale peer-to-peer networks is **churn**, the change-over of nodes joining and leaving the network. Within a large-scale peer-to-peer network there is, in practise, a turnover of nodes. Those wishing to use the service will tend to connect and disconnect as soon as they are finished using the service. This is particularly true of file sharing services such as Kazaa and BitTorrent for which some nodes may leave the network after only a few minutes. Churn produces a strain on the network as rapidly connecting and disconnecting nodes may lead to broken links in the overlay network due to leaving obsolete data in the routing table of other nodes. This can cause failed lookup requests. (Rhea et al., 2004)

Attacks

Urdaneta et al. has performed a very detailed analysis on security techniques employed by several DHT algorithms to prevent against three kinds of common attacks on peer-to-peer networks: 1) the Sybil attack, 2) the Eclipse attack, and 3) the routing and storage attacks. (Urdaneta et al., 2011)

A Sybil attack exploits an inability in decentralised systems to confirm that a single node represents a single physical device. If this cannot be proven then it is possible for a malicious device to flood a DHT network with fake virtual nodes which can then influence the network. One proposition for resisting Sybil attacks is to implement a distributed registration system for nodes to authenticate themselves. This method is not Sybil-proof but rather Sybil- resistant. A node joining the network would need to register itself against registration nodes, providing its IP address and incoming port number and confirming that the hash it has generated for itself is valid; thus limiting the identities allowed from a single node. (Dinger and Hartenstein, 2006) This approach introduced new issues however, trust management algorithms are required to ensure that the registration nodes themselves are not malicious. Although Urdaneta notes many other solutions exist.

Another possible exploitation is the Eclipse attack. In a DHT network nearby nodes, called neighbours, may serve as the first hop for most outgoing, and final hop for most incoming, requests. The attack gets its name from the action it performs, if a node is *eclipsed* then it becomes obscured from the network; other nodes may be prevented from successfully locating it and this node may be prevented from contacting foreign nodes. An Eclipse attack is small- scale, it cannot be used easily against multiple nodes at once. A degree of herd-immunity is expected by a large-scale DHT network; while some nodes could be targeted specifically by such an attack, the majority of nodes would be able to continue operating unhindered.

An Eclipse attack can also be the preclude to a routing and storage attack, unlike Sybil and Eclipse, this type of attack directly disrupts the DHT algorithm. A malicious node may drop requests instead of forwarding them for instance, or respond to valid lookup requests with invalid data. Due to the redundant linking aspect of Kademlia 2.1.4 it is more resistant to this type of attack than Chord 2.1.2 or Pastry 2.1.3; since if a lookup request fails it can be re-routed through another node.

2.2 Relaying

2.2.1 Onion Routing

Onion routing is a method of data relaying which is the primary feature of the anonymity network TOR, The Onion Router. This novel method of anonymous data transmission was discussed by Reed et al. who states:

> "Most security concerns focus on preventing eavesdropping, i.e. outsiders listening in on electronic conversations. But encrypted messages can still be tracked, revealing who is talking to whom. This tracking is called traffic analysis and may reveal sensitive information."
>
> (Reed et al., 1998, p. 482)

Research into onion routing has instead focused on obfuscating traffic analysis techniques and preserving anonymity between connections.

The onion routing methodology builds upon that of *anonymous re-mailing* which is a technology used to hide the source of an e-mail address by using a proxy to strip the sender information and forward the e-mail. Onion routing improves these ideas in two important ways: one being that communications take place in real time, although additional encryption layers and network latency mean connections are not as fast; and two that communications are bi-directional, allowing for the use of two-way request-and-respond services like HTTP, VoIP, and SFTP to name a few possibilities.

Using onion routing, the sender and recipient of a data packet do not make direct contact with each other; instead the data is relayed between devices called *nodes* or *routers* that act like proxies. These devices maintain a constant socket connection between one another which are multiplexed, meaning used to transmit multiple messages at a time. (Reed et al., 1998)

Onion routing is reliant upon asymmetric key cryptography, routers make their public key known so that any device wishing to make use of them can ensure that their encrypted message can only be decrypted by them. (Goldschlag et al., 1999)

When the initiator wants to communicate with a responder, the initiator will determine a path (or *chain*) which is typically selected randomly from a pool of known routers. Once the chain has been established the message can then be cryptographically layered by successively encrypting the message with the public key of each router handling the connection from the destination to the source. Once the message is layered, it becomes known as an *onion*, and is routed onto the path. As the message is transmitted, each layer is striped off by each subsequent node until the last layer at which point the message has reached the recipient. The recipient can use the same procedure to respond.

The onion routing technique preserves anonymity by limiting the available knowledge about the sender and recipient as far as possible; anybody snooping on the network may detect a connection between two nodes, but not have any idea who sent the packet, what's in it, who it's from, or who it's to. (Haraty and Zantout, 2014) Using cryptography in this manner gives a distinct advantage over tunnelling methods like SSL and TLS where encryption is only maintained in transit. The chain is as strong as the strongest link, hence only one honest node is required to maintain the security and privacy of the data and route. (Reed et al., 1998)

Garlic Routing

Garlic Routing is a relaying technique that builds on-top of onion routing by introducing parallel messaging and redundancy. When a node sends a message using this method, the message is duplicated and built into multiple onions called *cloves*. This increases the chances that the message is received by the recipient; since it only requires one of the *n* cloves to reach the destination, which helps to resist blocking and packet loss. (Hooks and Miles, 2006)

Garlic Routing has been implemented by The Invisible Internet Project (I2P), a project that aims to provide an alternative to Tor. In this context it allows for *mixnet messaging* meaning that a *bulb* (terminology for onion in a garlic context) contains multiple requests for the end-point. Although the technology is already implemented in the I2P software, the I2P website points out that most of these bulbs typically only carry one message anyway, leaving any advantage over standard onion routing rather mute. (Invisible Internet Project, 2014)

Note that the style of garlic routing described by I2P differs from the style described by Hooks and Miles, as the objective of the implementation by I2P is not to add redundancy to the message sending but instead reduce the overhead of sending multiple messages.

2.2.2 Blocking

In computer networks, blocking is the process of a network controller preventing the transmission of data. In typical real-world examples this often refers to governments with control over Internet Service Providers preventing the transmission of certain types of traffic, but this is also applicable to smaller networks like that of an education institution; in the former case blocking is considered a form of censorship.

Data packets can be scanned and blocked with two different methods: shallow packet inspection, which only blocks based on the destination of a packet; or deep packet inspection, which reads the contents of a packet and then decides if it should be blocked. (Porter, 2005)

The garlic routing 2.2.1 technique can be one solution used to prevent blocking, by sending the same message along different routes, there exists a greater possibility that the packets will bypass filters en route. Filters at the source or destination however would use the same rules to filter packets, so this alone would not be enough to bypass those.

Multiple access points are generally the go-to solution for blocking resistance. It's common for countries to block access to TOR nodes for example, the TOR service has adapted to this by using bridge relays; nodes which act like proxies for access into the TOR network.

One of the more interesting methods of blocking resistance is implemented by the Infranet service; it provides nodes that appear to act as harmless web servers but use hidden content in allowed HTTP traffic to forward requests to their intended destination. This method does not provide anonymity. (Köpsell and Hillig, 2004)

Dust is described as an internet protocol designed to resist *known* methods of packet inspection and blocking. Dust uses full packet encryption to keep the contents of the message secure, and a variety of features such as message chaining, randomised padding; these features help to prevent against filtering techniques like string matching (searching for patterns in the packet data), length matching (blocking repeat packets that are the same length) and timing pattern matching (blocking packets that follow disallowed ones). (Wiley, 2011)

2.3 Anonymity

2.3.1 Anonymity Metrics

The term anonymity means 'to be anonymous', meaning to not be identified. In the information age, there's so much identifying data flying around the internet tubes that being anonymous is a challenge.

While the term anonymity has a well understood meaning, one of the challenging aspects of research in this area is determining how anonymity can be assured; the question to be asked is how do we measure anonymity?

Zhu and Bettati draws from prior research into the topic to provide a metric for anonymity that is adaptable for the *heterogeneity*, the diverse range of content, of more complex anonymity networks like Freenet and TOR. (Zhu and Bettati, 2005)

Zhu makes the case that the measure of the anonymity degree should comprise of at least these requirements, the anonymity degree metric must:

- Capture the quality of anonymity; entropy, the logarithmic rate of data transfer, has been shown to be a good metric for this.
- Consider the topology of the network; the topology will affect how any attack is performed as it affects the degrees of separation between users.
- Be independent of the size of the network; more users involved may improve anonymity but that does not guarantee the quality of anonymity is good.
- Be independent of the threat model; attackers will use a variety of techniques to breach anonymity in the system, the metric should not be specific to any one type of attack, it should be general.

2.3.2 Anonymising

The next issue to consider is the practicalities of how to be anonymous. For an anonymity network to be anonymous, the availability of identifying information must be minimised.

Any node in a distributed network faces the initial issue that its participation is noted by any device it comes into contact with; its presence is public. This isn't directly an issue from an anonymity stand-point but such presence may be considered suspect. In this instance there may well be a *safety in numbers* aspect, where more nodes make it difficult to differentiate who is who. TOR deals with this by allowing users to choose how they want to join the network: a device may act as an exit node, allowing connections to leave the onion network; a relay, allowing connections to onion-route inside the onion network; or do neither and not act as a router at all.

Note that references are made to data security at times instead of anonymity; ensuring that identifiable data is secure is one of the critical aspects of maintaining anonymity.

Octopus, Secure and Anonymous DHT

In addition to the Distributed Hash Tables mentioned in section 2.1, Octopus is a DHT protocol that claims to offer security and anonymity in addition to DHT functionality. Octopus is based on Chord 2.1.2 but includes additional mechanisms for discovering attackers and malicious nodes in the network. (Wang and Borisov, 2012)

One of the methods in Octopus used to aid anonymity is *dummy queries*. Attacks such as timing analysis may be used on nodes in an anonymity network to deduce which nodes are communicating by measure of when they transmit. Dummy queries obfuscate this data by sending *blank requests* to other nodes, thus making timing analysis attacks useless.

Another technique used to aid security of DHT requests is lookup redundancy which is a technique similar to garlic routing 2.2.1. Keys in the DHT are replicated several times in the network, then when a node performs a lookup to find the key, it sends several requests to the various nodes. The node performing the request will then receive several responses back. Ideally all these responses should be consistent, but if one node responds with the wrong answer then it can be assumed to be faulty or malicious, if multiple nodes respond with the wrong answer then this becomes more difficult.

Octopus also protects against routing and storage attacks by having each node maintain a list of predecessors in addition to successors. Nodes in an Octopus network can do checks on their predecessors to check that they are routing information correctly, if a predecessor to node X does not contain node X in its routing table, then that node is being a naughty node. Such requests can be sent via from other nodes too; malicious node Y may tell it's successor X that it is in the routing table, but tell random node A that X does not exist.

The problem Octopus has in the context of distributed systems is that its method of expunging bad nodes is reliant upon a centralised service. Nodes may only participate in the network if they have registered themselves with the centralised Certificate Authority that is responsible for verifying node IDs. The techniques used are good for detecting malicious nodes, but there must be some power structure for a decentralised system to remove any node from the network; these are the issues dealt with by trust management.

2.3.3 Asymmetric Cryptography

Asymmetric cryptography, also refereed to as public-private key cryptography, is a cryptography methodology where a plain-text message is encrypted and decrypted with two different paired keys. The typical example involves a person called Alice sending a message to a friend called Bob. Bob has his own public-private key pair, he keeps his private key a secret and gives Alice his public key. The theory is that when Alice encrypts her message to Bob with Bob's public key, only Bob is able to decrypt it. A third party, Theresa, may want to snoop on the data, but without the private key she cannot read it. Even Alice is unable to decrypt the message once it's encrypted. The chances of brute-force decrypting the message in a short time are incredibly remote. These characteristics make asymmetric cryptography incredibly useful for secure online communication.

The basis for the maths behind asymmetric cryptography is the ability for computers to solve difficult mathematical problems, ones which require a *brute force* approach to solve because no general solution exists. (Vassilev and Twizell, 2012) Vassilev et al. gives a description of three important asymmetric algorithms here.

Diffie-Hellman Key Exchange

The Diffie-Hellman (D-H) Key Exchange was first writen about in 1976. It relies on the problem of discrete logarithms, which have no known general solution. Keys between two participants in the exchange are determined on pre-agreed prime and integer numbers which are shared across an insecure network, these values are the basis for which participants generate their private keys. The primary use case for D-H is between two devices, although multiple devices can take part, this increases the complexity of the algorithm.

RSA (Rivest, Shamir & Adleman)

Another example of a public-key cryptosystem is RSA. The core mathematical problem it relies upon is factoring a very large composite number which is the product of two large prime numbers; there is no general solution for this problem hence factoring the large number requires guesswork, far more computing intensive than reasonably feasible. Unlike in the D-H key exchange, the keys generated by participants are unrelated to one another, no initial exchange is needed to generate the keys. RSA is commonly used in SSH connections. It was first used in 1978 and to date it is considered unbroken.

Elliptic Curve Cryptosystem

An elliptic curve is graph curve derived from the equation $y^2 = x^3+ax+b$. Its use in cryptography is more recent than D-H or RSA, having been first written about in 1985. Elliptic curves serve as the basis for algorithms such as the Elliptic Curve Diffie-Hellman Key-agreement protocol, taking the D-H system and applying the elliptic curve discrete logarithm problem to ensure a general solution is even harder. One of the main advantages it holds over simple D-H and RSA is that the encryption and decryption operations are faster. (Vassilev and Twizell, 2012)

Cryptosystem choices

Vassilev et al. states that most authors consider the RSA-based systems to be the most secure. (Vassilev and Twizell, 2012) I2P makes use of the D-H key exchange algorithm with a fairly long 2048 bit key. (Invisible Internet Project, nd) TOR currently implements its asymmetric cryptography with RSA-1024 (RSA with a 1024 bit key), the developers are looking to replace it with the newer ECC cryptosystem. The effect of Moore's Law means that brute-forcing these cryptosystems will become progressively easier; RSA is unbroken but RSA-1024 is considered *weak*. (Mathewson, 2010)

THREE

ANALYSIS: COMMENTS ON REQUIREMENTS SPECIFICATION

Following on from the analysis in chapter 2, the requirements specification lists the specific requirements that will be expected of the project prototype.

3.1 Functional Requirements

3.1.1 High-Level Functional Requirements

Requirements 1–5, subsection B.1.1.

The high-level functional requirements form the basis of the use cases of the system; these are necessary requirements that the prototype product must fulfil to meet the aims of the dissertation. They correspond to the work proposed in the Terms of Reference, section A.3. Note that each item builds on the previous, in this manner they roughly outline the order in which functionality will be developed. They will later form the requirements for integration testing during synthesis.

3.1.2 User Interface Requirements

Requirements 6–9, subsection B.1.2.

The focus of this project is not to build a flashy UI, so these requirements are basic. A user will interact with the software through the terminal alone; command line arguments will be the settings used for initialising the node. A set of commands will be provided for controlling the node, although a lot of the network maintenance should happen in the background without interrupting the user. Information and statistics may be printed to screen, though this will be for debugging and demonstration purposes.

3.1.3 Communication Requirements

Requirements 10–16, subsection B.1.3.

These requirements draw from what has been learned in the literature review. The imperatives are: the use of encryption, for security and anonymity; prompt and correct response to DHT information requests, as these are important to the healthy functioning of the network; and verification, basic verification of node information is possible by ensuring that the node provides a valid hash based on the IP it is connecting from. Padding is a technique that was identified for disrupting blocking.

3.2 Non-Functional Requirements

3.2.1 Data Structure Requirements

Requirements 17–19, subsection B.2.1.

Stemming from the analysis into DHT algorithms, it was noted that there is important data to be stored. What this data is, and how it is structured, is fairly similar across the three DHT algorithms discussed. For the product to be developed, the same problems need to be solved. Structures will be put in place for handling fingers, and a finger table store will be created.

The Terms of Reference stated that the analysis would inform on a decision of which DHT algorithm to implement. The analysis however has demonstrated a great deal of complexity in the implementations of these algorithms. The Kademlia DHT will serve as inspiration for the development although, as a prototype model, it is expected that the product implementation will not have the same complexity.

3.2.2 Network Requirements

Requirements 20–22, subsection B.2.2.

Nodes need to be able to identify themselves uniquely, this is achieved with a hashing algorithm, which is able to turn any sort of node data into a set-length bit pattern. The specific hashing algorithm to be used is dependent upon requirement #24.

As a prototype, dealing with network layering, techniques like Network Address Translation lie outside of the scope of this project. It is expected that this product will only function on an internal LAN, like that set up in the labs at the university.

When a node leaves the network, it informs other nodes so that its routing information can be removed from their routing tables. If a node fails and drops out of the network ungraciously, then it won't have opportunity to perform any pass-off procedures. A heart-beat functionality could be used to determine that nodes are still alive and well.

3.2.3 Security Requirements

Requirements 23–26, subsection B.2.3.

One of the most critical aspects to the anonymity networks discussed in the literature review is their public key cryptography. Having looked at some of the possible options for algorithms, it has been decided to use RSA encryption. It is still considered a secure algorithm for the task and the persistence of keys it offers in comparison to that of Diffie-Hellman make it a simpler algorithm to use too.

While SHA-1 is appropriate for use as a hashing algorithm; serving as the basis for node IDs in Chord, Pastry, and Kademlia; the analysis into hashing algorithms revealed that the weakness of SHA-1 could become a potential risk in future. SHA-2, with a 256 bit digest, makes a more secure and more collision resistant choice.

Padding is a technique that was identified as an anonymity and anti-blocking feature. By adding random data into the packets being encrypted and sent the length of data will be varied and bit-pattern matching made redundant, hence making blocking more difficult.

Additional blocking-resistance techniques could be included, like garlic routing and node registration, but these features may add additional complexity which is not achievable in the timescale.

3.2.4 Operating Requirements

Requirements 27–29, subsection B.2.4.

The system will be developed in a Linux environment, so it is expected to function there; however Python is portable so it may also work in any environment with a functioning interpreter.

Error tolerance is also included here. In a perfect world, all code always works and does what we want, however this is not a perfect world and the code will likely not be perfect either. The product needs to be error tolerant to the degree that encountering an error should not cause failure of the entire system. Instead, the program should attempt to recover from any errors as best it can.

FOUR

SYNTHESIS

This chapter discusses the design and implementation of the system, named DistrIM, that has been produced as part of this dissertation. Section 4.1 discusses the design of the system. Section 4.2 discusses the the protocol used for communication by nodes. Section 4.3 discusses the technical challenges faced during the implementation and how the system design has changed to accommodate. Section 4.4 sets out the testing plans which were used to assure correct functioning of the program.

The diagrams produced during the design portion of the project are available in full in Appendix C. Program documentation is available in Appendix E.

4.1 Design

The overall design of the system has been adapted in an agile manner as the development of the system has progressed. Simple design ideas gleamed from the requirements analysis in chapter 3 have been built upon throughout development to provide enhanced features.

4.1.1 Class Diagram

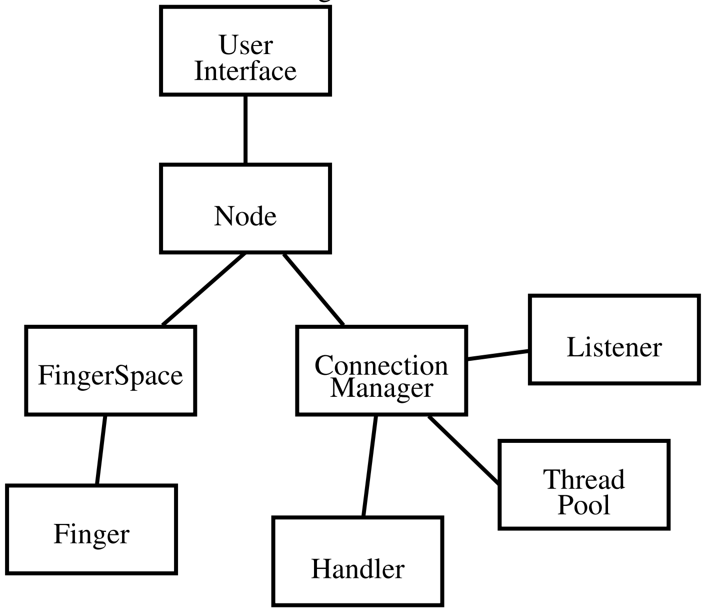
With the intention of implementing the final product in the object-oriented language of Python, it was deemed appropriate to first model the system in UML (Unified Modelling Language).

The final main class diagram in section C.1 was built up over several iterations to progressively delegate functionality, while trying to maintain high logical cohesion of classes with low coupling.

One of the first designs for the model of the system is shown in Figure 4.1. The lines between classes show weak associations at this stage of the design.

The classes of note in this diagram are `FingerSpace`, `Handler`, and `ConnectionsManager`. `FingerSpace` is the class responsible for storing fingers and providing the algorithms for node lookups. `Handler` is responsible for communication between nodes once an initial connection has been established. `ConnectionsManager` is responsible for all incoming and outgoing connections. The `Listener` class shown here had responsibility for accepting new connections, although having to call-back to the Connection Manager to process jobs made it unsuitable; it was determined that the functionality was shared and thus the methods of the Listener class were merged into the methods of the Connection Manager.

Fig. 4.1: Initial class models showing structure and association between classes.



4.1.2 Connection Handlers

The `Handler` class was broken down into a series of classes which inherit methods from a common `ConnectionHandler` class. The methods for communicating with nodes at the packaging and transmission level are the same, however the subclasses of `ConnectionHandler` have specific responsibilities and methods for the type of request being handled; this is dependent upon the message type, or verb, (defined in subsection 4.2.1) that initiates the connection.

The `IncomingConnection` E.3.4 class is the most feature-full; an instance of this class is initialised to handle an incoming connection from a foreign node. This class is the most broad in terms of methods, mostly because a specific handler type cannot be determined before data transfer begins, so all options must be accounted for.

The `MessageHandler` E.3.4 class is the part which implements the packaging part of the onion routing algorithm. When a node wants to communicate with another node, the sender initialises an instance of this class, populates it with information about the foreign node, and then makes a call to the SEND_MESSAGE method to initiate sending.

Descriptions of all handlers is available in subsection E.3.4.

4.1.3 FingerSpace, *the Finger Table*

`FingerSpace` is the class responsible for storing information about nodes, it represents the share of the Distributed Hash Table (DHT) of which this node is aware. Data is stored in a dictionary object, a hash map, which maps keys to values exactly the way a DHT does. The key is the IDENTIFIER and the value is the `Finger` object representing the node. The Finger class is discussed in subsection 4.1.4.

Whilst the system design has been designed to minimise coupling, the data it contains is required by almost all parts of the system, particularly the connection handlers. The `FingerSpace` object is instantiated once by the `Node` object and passed as a parameter to connection handlers from the `ConnectionsManager` class.

Because the application uses multiple threads (discussed in subsection 4.1.5) it was important to control access to prevent possible concurrency issues. To solve this potential problem, a *semaphore* was introduced which is held as a private attribute. Any object attempting to access the internal finger table through public methods will be blocked until the semaphore is available.

4.1.4 Finger, *identifying nodes*

As identified in the analysis, nodes in the DHT overlay network are represented with *fingers*. A finger is a structure that contains the information necessary to communicate with a single foreign node. The finger, like the node-specific attributes, are unique to the node they represent.

A class was designed, appropriately named `Finger`, to represent the foreign node attributes, which are:

IP ADDRESS The four-octet IPv4 address of the node device.

LISTENING PORT The port number on which the node has an open socket to listen for incoming connections.

PUBLIC KEY The public key of the node, used to encrypt messages intended for the recipient.

IDENTIFIER The node ID, this is calculated using a hashing algorithm that involves the other three attributes.

The finger class is described in section E.3.2.

Identifier, the Node ID

The identifier, or node ID, is determined by concatenating the IP address, listening port, and public key into a single string which is then hashed. The hash is calculated using the SHA-256 algorithm. As noted in the analysis the node ID must be unique within the network; it is considered an error if two distinct nodes share the same hash as it introduces ambiguity, making it likely that any messages may be incorrectly routed to the wrong node. As such, a good hashing algorithm should have a negligible chance of producing a collision.

Initially the hashing algorithm used was MD5, however SHA-256 is more collision resistant, due partly to its algorithm strength and also because SHA-256 hashes are twice the length of MD5 hashes. An MD5 hash is 128 bits long, giving 2^{128} possible values; however SHA-256 hashes are 256 bits long thus giving 2^{256} possible values, approximately 10^{77} more values. Having a fair few more values available makes the chances of encountering collisions negligible.

Designing the hashing algorithm in this way also helps to prevent against impersonation, one node pretending to be another. The stronger SHA-256 algorithm makes it harder to find alternative data to intentionally hash to a given value; since that value is requested when establishing a connection between two nodes, and the foreign node will test the hash validity, this method of attack becomes unfeasible.

For demonstration purposes the identifier is shortened to the first 16 bits of the hash. For a true large scale system such a short identifier would be inappropriate due to the risk of collisions being much higher.

4.1.5 Multi-threading & Concurrency

Nodes in peer-to-peer networks are expected to work as clients and servers, dealing with requests from foreign nodes whilst still being able to make requests on behalf of the user. For this reason, concurrency has been introduced into the application.

Concurrent operations take place in the `ConnectionsManager` class (section E.3.1). Two additional threads are created which, when the START method is called, begin running in the background as daemons. One thread, the listening thread, pends on the accept method of the listening socket object such that when a node requests a connection, it is handled immediately. This method, compared to polling for connections, wastes less CPU time and ensures prompt response to requests.

The `ConnectionsManager` also has a `ThreadPool`, which is a queue of worker threads which host Connection Handlers. The sockets formed when nodes connect are passed to these workers. Using this style of multi-threading allows multiple nodes to be connected simultaneously, a very useful feature for a distributed peer-to-peer network.

4.2 Protocol

As a networking application, the protocol was one of the most important considerations of the project. Sequence diagrams showing communications in the network are in section C.2.

4.2.1 Verbs, *actions*

The verbs define actions in the protocol; when one node connects to another, the first message type received by the listening node what type of connection handler to use, and thus the procedure that must be followed. If a message type is sent other than the expected message then a PROCEDURE ERROR exception is raised and the connection is terminated. Strictly defining the procedures in the protocol this way limits the potential, not only for exploitation, but also for errors. System errors could become the result of vaguely-defined procedures; such as timeout errors or receiving unexpected data.

The verbs used in the DistrIM protocol are defined as such:

Announce For a new node that has just bootstrapped into the network, via an existing node, to introduce itself to other existing nodes. The new node will send a copy of its finger to the foreign node.

Message For direct messaging, a node can contact another node directly but this functionality is obsoleted by the **Relay** feature.

Quit The opposite of **Announce**, if a node leaves the network cleanly (that is, if it doesn't crash) then the node informs all foreign nodes that it is departing. Provisions can be made if this is the case, for instance to remove the node from the Finger Table of foreign nodes and to update routing information if necessary. This is shown in subsection C.2.3.

Relay Part of the messaging and onion routing protocol described in subsection 4.2.5, there are two possible outcomes to a relay request, either a package of data is obtained which must be routed to another node, or a packet of data is obtained which contains a message for the receiving node. In the former case, the data is forwarded with the same verb to the next node.

Welcome Part of the bootstrap and rendezvous procedure described in subsection 4.2.2, when a new node connects to an existing node to join the network, it sends its own finger information. The new node expects to receive this welcome message back, congratulating it on successfully connecting to the network. The Welcome message will also carry the Finger Table of the existing node so that the New Node can announce itself.

Verbs are defined as class attributes, shown in section E.3.4.

4.2.2 Bootstrap & Rendezvous

When a node is created, it initially has no knowledge of any other node on the network. A process called bootstrapping is performed where the new node attempts to establish a connection with a node already in the network. To do this, the new node must already be aware of the IP address and listening port of an existing node. Once the initial connection is made, the existing node will share a copy of its Finger Table with the new node. The new node will rendezvous with other nodes in the network by announcing itself to the node listed in the Finger Table of the existing node.

Without knowing the public key of the existing node, the new node is unable to properly communicate with the existing node following the network protocol. Instead the new node must make it possible for the existing node to send back protocol messages. The new node will send, unencrypted, the contents of its own finger.

This communication is the only message in the protocol which is not encrypted. However the data transferred is part of the publicly available DHT, so no sensitive data is exposed.

This process is shown in subsection C.2.1.

4.2.3 Message Structure

The basic structure of messages follow the format:

[MESSAGE_TYPE, PARAMETERS]

Where the MESSAGE_TYPE refers to one of the verbs defined in the protocol description, subsection 4.2.1; the PARAMETERS contains the information to be communicated, these must be relevant to the verb. Parameters are formatted as a hash-map, a dictionary in Python, like so:

$$\text{PARAMETERS: } [(\text{KEY}_1 \rightarrow \text{VALUE}_1), \ldots, (\text{KEY}_n \rightarrow \text{VALUE}_n)]$$

Messages are also sent with two other important pieces of data: the finger of the local node, and some randomly generated padding. The local finger information can be used for verification by the foreign node and can enable the foreign node to communicate back to the local node in case the foreign node was previously unaware of the local node. The padding adds cryptographic strength to the message by disguising the contents of the encrypted packet, it is discarded when the message is decrypted.

The unencrypted message structure appears as such:

Message Structure			
Local Finger	Message Type	Parameters	Padding

The message, structured as a length-four tuple, is serialised (or Pickled in Python terminology) into a string representation. The string representation of the message structure can be encrypted easily for transmission.

The RSA encryption algorithm (subsection 4.2.4) used to secure the data has a limitation on the length of data that can be encrypted at any one time. The problem, discussed in subsection 4.3.1, was fixed by encrypting the serialised data in chunks and rebuilding those chunks for de-serialisation on the other side.

The structure of the message during the transmission stage, when the byte-steam is sent through the socket to the foreign node, is as such:

Transmission Structure				
Data Length *4 bytes*	Block 1 *128 bytes*	Block 2 *128 bytes*	Block 3 *128 bytes*	Block *n* *128 bytes*

The encrypted message is made up of the encrypted blocks prefixed with the length of the byte-stream. The data length is used to verify that all data has been received before attempting to decipher the information.

4.2.4 Encryption

RSA, being a well known standard of encryption, was a suitable choice for encryption between nodes. Messages encrypted with RSA public-keys are considered unbreakable for practical purposes.

Public-Private encryption provides a useful feature which is that when a plain-text message is encrypted with a public-key, the only way it can be decrypted is with the private key. The public key is part of a node's Finger, and such it is available to any node that has the finger. The private key is created by the `Node` class and passed to the connection handlers, the private key is never transmitted outside of the system and is considered secure.

Public-Private key encryption is slow compared to other forms of encryption, such as symmetric encryption; however the focus of this project focuses on the security and as such speed was not the primary consideration.

An alternative method which was considered was using fast symmetric encryption, such as AES, to encrypt the data and then encrypting the cipher key with the RSA key. This may be quicker than decrypting large payloads with RSA alone, although the additional complexity was considered surplus to requirements.

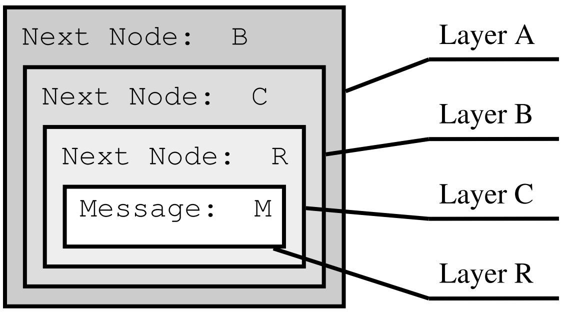
4.2.5 Onion Routing, Relaying

One of the major features discussed in the analysis is onion routing, a relaying method where a message is bounced between a series of randomly-selected nodes in order to disguise the true source and destination of a message. The term onion in this method name refers to the way in which the message is *layered* and how those layers are *peeled* away during transmission. Transmission is shown in the sequence diagram in subsection C.2.2

The way this method works is that the message from the sender (Node S) is encrypted with the public key of the intended recipient (Node R). With the public-private key encryption algorithm used there is a reasonable guarantee that the message, regardless of where it is relayed to, can only be read by the recipient R. At this point, the message data is considered *packaged*, into the first layer.

The encrypted package is then repackaged and encrypted with the public key of each node in reverse order of the path traversed; so for a given example where the message M is sent from S to R via nodes A, B, and C, the message M is first encrypted with the public key of R, then the public key of C, then that of B, and finally A. The data must also be packaged with path information, the NEXT NODE, so that the current node knows where to relay the data. Figure 4.2 shows the structure of the relay package constructed by Node S.

Fig. 4.2: Demonstration of the packaging of MESSAGE M for transport along a route of three relay nodes: A, B, and C.



Each successive node in the path will have to peel a layer of encryption away, the instructions decoded by *peeling* that layer inform that node what to do with the next layers in the package. If the decrypted package has the NEXT NODE attribute then the package should be forwarded to the specified node. If the decrypted package has the MESSAGE attribute then the current node is the recipient; thus the message has been delivered and it will be displayed to the user at that node.

In this case the SENDER attribute allows the recipient to know from which node the message originated.

4.3 Implementation

Some of the utility classes and functions (subsection E.4.2) have been developed due to requirements uncovered during implementation; those problems and other issues of implementation are discussed here.

4.3.1 RSA Encryption

Encryption functionality in DistrIM used is provided by the PyCrypto library.

One issue which was encountered after implementing the RSA encryption was that the PyCrypto function has a limitation, which is that only 128 bytes can be encrypted at a time. Some package loads are greater than 128 bytes, and when this was the case no error was raised locally; after transmission the foreign node would attempt to decrypt the data and get garbage. The solution to this problem was to implement a chunking algorithm, defined in section E.4.2, which split the serialised unencrypted data into chunks of a set length (128 bytes in this case).

The chunking algorithm seemed to solve the problem at first, however the reliability of the system was noticeably poor. Transmission of messages would often lead to errors during de-serialising. After a diagnostic of the data, comparing the MD5 hash values at the stages, it was determined that the message was becoming erroneous after decryption; despite the MD5 hashes of the data before decryption matching. It was eventually realised that the logic flaw was assuming that encrypting 128 bytes would return 128 bytes; the RSA algorithm would often return 127 bytes instead, moving the position of future chunks such that decrypting them would produce garbage.

The solution to this second problem was to make a structure for encrypted data; the CipherWrap class E.4.2, which provides ENCRYPT and DECRYPT methods that wrap around the PyCrypto library, was made to place chunks in a list and serialise them.

Unit tests were in place to test this part of the code which were passing, unfortunately the test data used did not stress the program enough to uncover the errors, which is why the problem persisted for so long.

4.3.2 Concurrency

Concurrency is notoriously difficult aspect of computing to use correctly, and it has a source of several implementation issues during development.

One of the first iterations of the `ConnectionsManager` simply had the listening socket in a separate thread which simultaneously did the connection handling. It was occasionally a problem that, if an exception occurred, then the listening thread would halt. To fix this, broad exception handling was put in place to prevent uncaught exceptions from killing the thread.

Another concurrency issue appeared later during unit testing of the `FingerSpace` class. The PUT method for inputting singular node information made use of a SEMAPHORE to prevent interference. A new method was included in the class for importing a batch of node data, this method acquired the semaphore and then called the put method; this resulted in the thread trying to re-acquire a semaphore that it already held, thus entering deadlock. The method was re-written to avoid fetching the semaphore twice.

4.4 Testing

The testing strategy is composed of two parts: unit tests and integration tests. Unit Tests have been created during development to demonstrate that individual components of the product are functional. Integration tests will demonstrate the functioning of the system as a whole.

4.4.1 Unit Tests

Unit Tests are a very useful part of the development process, several issues in functions and class methods have been identified because of unit tests.

A test-driven development method was adopted for smaller parts of the system. The utilities E.4.2 for example contain small component functions and classes which can be easily tested independently from the rest of the product. Using the test-driven approach, the tests for functions were written first; knowing what the input is and asserting that the output is what we expect it was possible to then write the production code.

Unit tests should not only test correct handling of valid input, but also test the correct handling of invalid input. Several of the unit test blocks include TEST_VALID and TEST_INVALID block to show that program parts respond as expected to valid and invalid data.

A range of test data has been used, particularly for the protocol E.3.4 module; a generated file of test node data was used with many nodes and repeated tests to be robust and stress-test the system well.

Unit tests are shown in section F.3. There are 47 unit tests in total which all pass.

4.4.2 Integration Tests

The integration tests are intended to show correct functioning of the system as a whole. The integration tests used are based on the common use cases of the system. To pass, they should produce the expected result.

1. **Initialisation Test**

   Test Parameters Run the program to initialise an instance of a node, without any parameters.

   Expected Result The node should start, raising no errors, and display a message stating that it is ready for incoming connections.

2. **Bootstrap, initial connection**

   Test Parameters Run the program to initialise an instance of a node, then load a second instance of the program with instructions to connect to the first node.

   Expected Result Both nodes should start without errors, the bootstrap procedure should take place. Both nodes will have the finger of the other node in their own finger table.

3. **Announcing and rendezvous**

   Test Parameters Run the program to initialise an instance of a node A, then load two more instances creating nodes B and C, both with instructions to connect to node A.

   Expected Result Node A should have the fingers of nodes B and C in its Finger Table. Node B should have the fingers of nodes A and C in its Finger Table. Node C should have the fingers of nodes A and B in its Finger Table.

4. **Departure, maintaining integrity**

   Test Parameters Run the program to initialise an instance of a node A, then load two more instances creating nodes B and C, both with instructions to connect to node A. Node A will then disconnect gracefully.

   Expected Result All nodes will have fingers of all other nodes; when A disconnects, nodes B and C will still be aware of one another.

5. **Single Message Passing**

   Test Parameters Initialise two nodes, A and B. Connect node B to node A. Send a message from node A to node B, then send a message from node B to node A.

   Expected Result The nodes become aware of each other at the bootstrap stage. Simple message passing occurs without error, the test message from each node should be displayed on the opposite node in full without corruption.

6. **Message Passing via Onion Routing**

   Test Parameters Initialise two nodes, A and B, followed by 5 more nodes, which bootstrap against any node in the network. Send a message from node A to node B.

   Expected Result All nodes should be able to connect and become aware of all other nodes without error. Upon using the send message command from node A, a random path of nodes should be selected. Those nodes should note that they have relayed a message. Node B will receive the message in full without corruption.

FIVE

EVALUATION

5.1 Product Evaluation

The product, the software DistrIM that has been developed and its deliverables, largely meets the requirements that were set out for it in the Terms of Reference, Appendix A. Though some compromises have been made in order to meet time restraints, the core functionality of the product is in place.

5.1.1 Fitness for Purpose

The idea for the project was initially spawned from the desire to produce an application capable of anonymous instant messaging. This idea was fleshed out to cover anonymity networks and peer-to-peer networking. The core functionality expected of the product has always been secure messaging, and this has been achieved.

The requirements specification in Appendix B gives more specific requirements for the system to achieve. The first five requirements are the high-level functional requirements which mirror the use cases of the system. Test plans in section 4.4 were drawn up to show the functioning of the system and hence ensure that these requirements would be met. The results of testing are available in Appendix D, where the six integration tests are shown having been performed and passed as expected.

Several of the requirements make reference to DHT algorithms, and requirement #19 makes a specific reference to the data structure of Kademlia. One of the intentions of this project was to build a structured peer-to-peer network, based on a DHT algorithm as identified by the literature review research;uUnfortunately this has not been fully achieved. The produced product is a distributed system as it is peer-to-peer, however it is not structured like any of the researched DHT algorithms in section 2.1. Part of the reason for this is related to timing issues which are discussed below in section 5.1. While this is disappointing, it is not catastrophic to the functionality of the product; the system will work, but will not scale up to operate with the large numbers of nodes that are expected of networks based on Chord, Pastry, or Kademlia.

Despite the structural DHT omission, the research into DHT algorithms has identified requirements needed for distributed systems as a whole. Specifically the data structure requirements #17 and #18, which identified the need for some of the classes in section 4.1.

Network requirements too have been worked on, requirement #20 means nodes generate their own identifiers by hashing their local node data, and furthermore requirement #24 means this is done so in a way that is safe for use in a network. Requirement #21 has been fulfilled and will further be demonstrated at the viva. Requirement #22 was attempted, but not finished; PING and PONG verbs were initially a part of the defined protocol, but errors during implementation meant this functionality was abandoned.

Requirements pertaining to communication and security have been met. The finished product makes use of encryption for all communication (except bootstrapping, discussed in subsection 4.2.2). Integration tests demonstrate the responsiveness of nodes in the network, requests for data relaying can be dealt with promptly. Unit tests of the protocol handlers and utility wrappers also test valid and invalid data, confirming the fulfilment of these requirements.

Security requirements, some of the more mission-critical requirements of this product, have been fulfilled. The analysis of anonymity networks informed the design choice to use asymmetric public-key cryptography (requirement #23), deeper research into cryptosystems influenced the decision to implement RSA as the cryptosystem. Similarly, research into various popular hashing algorithms informed the decision of which hashing algorithm to use, culminating in the choice of SHA-2.

Requirements #24 and #25 relate to security and anonymity of transmitted messages. Analysis of anonymity networks discussed blocking as a possible impediment to the working of the network. Padding was considered and implemented, meeting requirement #24. Requirement #25 was, with hindsight, incredibly vague; it may be considered met based on the design of message transmission avoiding some of the types of packet inspection identified in subsection 2.2.2.

Overall, most of the requirements have been fulfilled. The most critical requirements, security and high-level functioning, have been met. Of the 16 defined *must* requirements, all have been fulfilled although the problem with the DHT structure leaves #11 only partially fulfilled. 6 of the 7 *should* requirements are fulfilled, again with the exception being #19. 2 of the four *could* requirements have been fulfilled.

5.1.2 Build Quality

The Synthesis chapter 4 gives some detail of the design and development process, including the agile design methodology, which is an important aspect of how the product has been created. The process is iterative, and as the project continued the design has been modified and expanded to include more of the features necessary to fulfil the requirements.

The requirements of the system were identified initially from the Terms of Reference, and adapted to include new requirements established from the Literature Review 2. The second part of the analysis, chapter 3, gives commentary on the requirements specification which provided the basis for the design decisions. The requirements remained consistent throughout design and implementation, they have not changed much.

The iterative approach has allowed for the gradual refinement of the product and the product deliverables. The design documentation supplied in Appendix C has started out from sketches and been redone to better delegate functionality between classes, keeping the program as modular as possible. the final version are those in the appendix. The UML diagrams provided give a good overview of the design of the final system.

The Terms of Reference warned that the code base for the product could become vast, fortunately good development practise has prevented any great messes being made. One of the major advantages of developing the product in Python is that the language is very easy to read and understand.

PEP8 is a style guide for Python code, it has been followed as closely as possible to minimise lint errors and help make the code as understandable as possible. Strict rules governing indentation and code style ensure that the code is easily read and understood. Where code is complex, adequate commenting has been included.

One other interesting approach to the development process was the use of test-driven development. Briefly discussed in section 4.4, this approach to implementation was adopted for the creation of smaller parts of the system such as the utilities module. By writing the unit test first, the developer is forced to consider the inputs and outputs of functions more; this methodology paired very well with the agile development approach. The result of this has been well-tested reliable code.

5.2 Personal & Process Evaluation

This project has been challenging; without doubt it has been the most difficult and substantive piece of work undertaken by the author, but as a result the most rewarding.

A great deal has been learnt about peer-to-peer networks and anonymity networks as a result of this project. At the start of the project, the author was aware of peer-to-peer networking as a technology but knew nothing specific about it. Learning about the types of peer-to-peer networks, and how structured DHT networks operate, has given a greater appreciation for the depth of research into this area. Aspects of structured peer-to-peer networks such as balancing, hashing, attacks, and trust management were unknown to the author prior to beginning the project.

The author is also now aware of the challenges faced by anonymity networks trying to keep content secure and anonymous. It is apparent that as much research goes into breaking anonymity networks as goes into hardening them—understanding how these networks can be broken is the key to ensuring they remain anonymous.

The Terms of Reference noted the possibility of the code becoming vast. The code for this project has been successfully managed through version control. Existing familiarity with the software Git made it simple to start using. Note that Git is intended to be used by multiple participants, the branching features have been immensely helpful for trying out new ideas without polluting the current working tree. Git, and use of the platform Github, have also helped serve as backup, in case of failure in some way the project would be backed up.

Managing of this project has allowed for the furthering of knowledge in subject areas for which the author did not posses much familiarity. Networking itself is a rather broad section of computing, and development of this product has required further learning of networking skills.

Time management has been one particularly difficult aspect of the project. A project plan was made as part of the Terms of Reference A.11, but unfortunately the deadlines of the project have not been adhered to. While it has been enjoyable to be able to plan ones own project and dictate the direction of work, the open-ended aspect of the project has been a blessing and a curse. These issues have highlighted a personal need to improve project management skills such as action planning and initiative.

But perhaps one of the key positives of this project is that it has bound together the knowledge and skills learnt over the past four years of the course; In particular those software development and design skills learn in previous analysis and design modules. All these skills have cumulated in a demonstration of not just the effort put in this year but the full four years of work.

SIX

CONCLUSIONS

6.1 Evaluation of Aims and Objectives

Aims

1. **To investigate how peer-to-peer anonymity networks protect the information shared by its users.**

   This aim has been met. Initial research into anonymity networks led quickly into research of distributed systems and peer-to-peer networks. The Analysis section gives a thorough description of peer-to-peer networks and describes the challenges present in maintaining them. The Analysis section also gives an overview of the technologies involved, such as hashing and encryption.

   The investigative aspect shows that peer-to-peer systems suffer from inherent security issues as a result of their open nature; without centralised trusted services, trusting nodes becomes difficult. However it has been shown that anonymity networks like Tor and anonymity-focused DHT algorithms like Octopus do give some functionality that can be used to preserve anonymity. The method of onion routing in particular can be used to transmit messages securely.

2. **To build a prototype instant-messaging application in an anonymity network, allowing users to securely share messages.**

   This aim has also been met. The product that has been produced is a prototype for an anonymous messaging system. Despite the lack of structured DHT, a distributed peer-to-peer network can be formed that connects nodes together. DHT is still used, when new nodes connect they will become a part of it.

   The use of cryptography that was investigated during analysis contributes to the anonymity of the system overall.

   The successful implementation of an onion routing algorithm satisfies the need for users to be able to share messages between one another. This feature provides the anonymity that is crucial to the aim of this dissertation.

Objectives

- **Research implementations of peer-to-peer networks.**

  Complete: The analysis gives an overview of three common DHT protocols, Chord, Pastry, and Kademlia.

- **Research techniques for maintaining anonymity.**

  Complete: Onion routing and garlic routing techniques have been identified, additional research into anonymous DHT Octopus gives notice of more anonymity techniques.

- **Analyse DHT algorithms and to determine an appropriate one for the network.**

  Complete: The analysis extracts information from the three DHT algorithms from which requirements are then specified. A decision was reached on which algorithm to implement.

- **Create design documentation, defining the protocol for sharing messages.**

  Complete: UML documentation shows the structure of classes of the program and the sequence diagrams demonstrate the protocol. Additional information about the protocol is in subsection 4.2.1.

- **Implement a DHT algorithm to construct a peer-to-peer network.**

  Incomplete: A peer-to-peer DHT network was created, but it is not a structured peer-to-peer network as was intended.

- **Implement a relay protocol to secure connections between nodes.**

  Complete: An onion-routing algorithm and protocol have been designed, implemented and successfully tested.

- **Implement anonymity features.**

  Part Complete: Package padding was one of the identified features, although additonal anonymity and security techniques had been identified in the analysis, they have not been used.

- **Successfully connect nodes together in a network.**

  Complete: This functionality has been demonstrated to work correctly.

- **Demonstrate a working network by showing nodes connecting to one another and transferring messages.**

  Complete: The integration tests demonstrate that this happens.

- **Evaluate the success of the network.**

  Complete: The evaluation discusses the success of the product.

6.2 Concluding Summary

As noted prior, the two aims of the project as defined in the Terms of Reference, section A.4, have been met. In addition, of the ten objectives defined, eight objectives are completed. Most of the aspects of the project have been completed as intended. The functional requirements of the implementation are mostly in place too, so overall the product is suitable and fit for purpose.

An aspect of the project that could be improved is the evaluation itself. Basic test cases, composed of unit tests and integration tests, have all the responsibility for showing that the system works. These tests alone may be adequate for functional requirements but security requirements would require more thorough testing. Some ideas for this are discussed in the future considerations, including attack models and anonymity metrics.

Overall, the project has been a success. A great deal has been learnt from the project analysis, and that knowledge has been used well during the implementation. Meeting the aims and most of the objectives are a good indication that the right approach was taken to development.

When this project was initially considered, the author's knowledge of peer-to-peer networks was limited to the very basics. It was not realised at the time that peer-to-peer networks themselves are such a large topic. As an area of study, it's provided many interesting problems, although only some have been directly

The original idea was to build a peer-to-peer network and add security features on top, however it become clear during analysis that P2P networks are not so trivial. The research performed is fairly general considering the scope of the subject, yet it leaves many stones unturned.

Peer-to-peer networking is a large area, and this product could serve as the basis for further research in the future.

The project, in successes and failures, has been an excellent opportunity to improve skills. Particularly research skills that have most certainly never been used as thoroughly as they have before.

6.3 Future Considerations

One of the identified facets of DHT algorithms that has not been expanded upon in this project is trust management. Within a P2P network it is considered the case that some nodes are malicious and cannot be trusted to provide valid data. Trust management is an area of research that seeks to provide ways of determining which nodes are honest and which nodes are malicious. Trust is a very difficult problem to solve in a peer-to-peer network as no one peer has more power than any other. However by incorporating a trust metric, it would be possible to hold elections in the network to delegate additional responsibility to certain nodes. For example, the distributed registration system for nodes proposed by Urdaneta et al. for protection against Sybil attacks could be implemented with firm reliability.

As identified, one of the issues with this project was the fact that no structured DHT was implemented. Further work into this topic could focus on expanding the ability of the product to handle larger numbers of nodes.

As an idea that was seen briefly around the Literature Review, one method of demonstrating the security requirements of the system would be to build an attack model. This would model the ways in which the system could be attacked or disrupted.

Another improvement to the evaluation would be to make use of the analysis on anonymity metrics. It was considered too difficult to use, and thus ignored during this project, but anonymity metrics would be very useful to have as a feature in any security-conscious network. It would give computists a way to prove the anonymity of their designs.

REFERENCES

Cheng-Zhong, X. (2005). *Scalable and Secure Internet Services and Architecture*. Chapman & Hall/CRC, Taylor & Francis Group.

Dinger, J. and Hartenstein, H. (2006). Defending the sybil attack in p2p networks: Taxonomy, challenges, and a proposal for self-registration. In *Availability, Reliability and Security, 2006. ARES 2006. The First International Conference on*, pages 8–pp. IEEE.

Goldschlag, D., Reed, M., and Syverson, P. (1999). Onion routing. *Communications of the ACM*, 42(2):39–41.

Haraty, R. and Zantout, B. (2014). The tor data communication system. *Journal Of Communications And Networks*, 16(4):415–420.

Hooks, M. and Miles, J. (2006). Onion routing and online anonymity. *Final paper for CS182S, Department of Computer Science, Duke University, Durham, NC, USA*.

Invisible Internet Project (2014). Garlic Routing and "Garlic" Terminology. `https://geti2p.net/en/docs/how/garlic-routing`. Accessed: 2015-04-18.

Invisible Internet Project (no date—n.d.). Introducing i2p, cryptography. `https://geti2p.net/en/docs/how/tech-intro#op.crypto`. Accessed: 2015-04-19.

Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., and Lewin, D. (1997). Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, New York, NY, USA. ACM.

Köpsell, S. and Hillig, U. (2004). How to achieve blocking resistance for existing systems enabling anonymous web surfing. In *Proceedings of the 2004 ACM workshop on Privacy in the electronic society*, pages 47–58. ACM.

Mansfield-Devine, S. (2014). Tor under attack. *Computer Fraud & Security*, 2014(8):15.

Mathewson, N. (2010). Initial thoughts on migrating tor to new cryptography. `https://gitweb.torproject.org/torspec.git/tree/proposals/ideas/xxx-crypto-migration.txt`. Accessed: 2015-04-18.

Maymounkov, P. and Mazieres, D. (2002). *Kademlia: A peer-to-peer information system based on the XOR metric*, volume 2429, pages 53–65. Springer-Verlag Berlin, Berlin.

Porter, T. (2005). The perils of deep packet inspection. *Security Focus*.

Reed, M. G., Syverson, P. F., and Goldschlag, D. M. (1998). Anonymous connections and onion routing. *IEEE Journal on Selected Areas in Communications*, 16(4):482–494.

Rhea, S., Geels, D., Roscoe, T., and Kubiatowicz, J. (2004). Handling churn in a dht. In *Proceedings of the USENIX Annual Technical Conference*, pages 127–140. Boston, MA, USA.

Rivest, R. (1992). The md5 message-digest algorithm.

Rowstron, A. and Druschel, P. (2001). Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001*, pages 329–350. Springer.

Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. (2001). Chord, a scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160. Date revised - 2007-02-01; Last updated - 2011-11-12.

Urdaneta, G., Pierre, G., and Steen, M. (2011). A survey of dht security techniques. *ACM Computing Surveys (CSUR)*, 43(2):1–49. http://dl.acm.org/citation.cfm?id=1883615.

Vassilev, T. S. and Twizell, A. (2012). Cryptography: A comparison of public key systems. *Algorithms Research*, 1(5):31–42.

Vu, Q. H., Ooi, B. C., and Lupu, M. (2010). *Peer-to-peer computing: principles and applications*. Springer, Berlin.

Wang, Q. and Borisov, N. (2012). Octopus: A secure and anonymous dht lookup. In *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*, pages 325–334. IEEE.

Wang, X., Yin, Y. L., and Yu, H. (2005). Finding collisions in the full sha-1. In *Advances in Cryptology–CRYPTO 2005*, pages 17–36. Springer.

Wiley, B. (2011). Dust: A blocking-resistant internet transport protocol. *Technical report*. `http://blanu.net/Dust.pdf`.

Xie, T., Liu, F., and Feng, D. (2013). Fast collision attack on md5. *IACR Cryptology ePrint Archive*, 2013:170.

Zhu, Y. and Bettati, R. (2005). Anonymity vs. information leakage in anonymity systems. In *Distributed Computing Systems, 2005. ICDCS 2005. Proceedings. 25th IEEE International Conference on*, pages 514–524. IEEE.

APPENDIX

A

TERMS OF REFERENCE

A.1 Project Title

Secure messaging within a decentralised anonymity network.

A.2 Background to Project

The focus of this project is on anonymity within peer-to-peer networks; this will involve research into methods of peer-to-peer networking, and showing how security and anonymity can be maintained in a peer-to-peer network. A basic messaging system will be created to demonstrate the network.

At this point in time, there are more devices and users connected to the internet than there have ever been. As a result more applications are dependent upon access to services on the internet. However we now know, as a result of the leaks by whistle-blower Edward Snowden, that internet traffic is widely snooped on by governments. In addition, we know that in some countries access to online services can be severed completely to enact censorship, as was seen during the Arab Spring when the Egyptian government blocked access to Facebook and Twitter. One area of computing that seeks to remedy these issues is anonymity networks.

An anonymity network can be defined as a system in which data is routed between nodes with the intention of hiding the true source of a data packet. Anonymity networks are based on peer-to-peer architecture. The best known example of such a network is Tor, The Onion Router.

A peer-to-peer (or P2P) network is a system of networked computers, referred to as nodes, that share information by direct connections to one another. In these systems the clients also act as servers that respond to requests for information from each other such that resources and services can be shared without the need for centralised servers.

In the typical architecture design of an internet service, a central server provides the service to which a number of clients can connect. The issue with this architecture is that the central server is a single point of failure, and a single point of attack; any weakness in the central server compromises the entire system. For this reason, anonymity networks must avoid centralised architecture as much as possible.

The P2P network architecture has been adopted by many services and protocols. Some of the most well known examples of the architecture include the likes of the music sharing application Napster, the file sharing protocol BitTorrent, and formerly the Voice over IP application Skype.

There are advantages to P2P networks compared with their traditional server- client counterparts. By aggregating the resources of multiple nodes, the need for costly centralised servers is removed; this also allows for P2P networks to be scalable and work with vast numbers of nodes, with many continually joining and leaving the leaving the network. P2P networks are also more robust and fault tolerant as data transferred around the network can usually be verified against multiple nodes. (Vu et al., 2010, p. 1–3)

Tor, The Onion Router, was initially developed by the US Navy. Tor is currently available as open source software supported by an organisation called The Tor Project. Tor was developed to allow users to send and receive internet traffic while remaining anonymous, this also includes protection against traffic analysis and eavesdropping. Part of the reason for the success of Tor is an anonymity technique called onion routing which allows the sender and receiver of a packet to remain anonymous. (Haraty and Zantout, 2014) Some preventable weaknesses do exist with Tor, such as exit node sniffing where data is intercepted as it leaves the Tor network through an exit node on it's way to an end user; this weakness can be solved if the end user's device is a node in the network, or if an encrypted protocol like HTTPS is used. Despite this, the Tor network as a whole remains intact despite efforts to de-anonymise it. (Mansfield-Devine, 2014)

This project will investigate methods of creating peer-to-peer networks and applicable security features such as encryption and onion routing. To do this, researched methods of P2P networking and security will be implemented into a basic P2P messaging application.

One aspect of this project will investigate the use of a Distributed Hash Table, or DHT. There exist multiple DHT algorithms which are used in P2P networks to allow nodes to discover one another. (Vu et al., 2010, p. 14) This will also involve comparing DHT algorithms to determine the most appropriate. This project will also look at the vulnerabilities of DHT algorithms against such attacks as Sybil, Eclipse, and routing and storage attacks. (Urdaneta et al., 2011)

Communications within the network must be secure, such that any message sent on the network can only be read by the intended recipient. Communications within the network must also be anonymous, such that it cannot be determined who a node is communicating with. It is not a requirement that users remain anonymous to each other, as users need to know who they are communicating with—only the sender and recipient should be aware of the source and end-point of any message sent through the network.

As we know, anonymity networks are often a problem for governments that wish to monitor or control their citizen's internet traffic. It is possible in a physical network to perform deep packet inspection and look at the contents of a packet, from this an authority can decide whether to allow the data to pass through or block it, preventing communication. A good anonymity network should have security features to prevent blocking, that is, it should be difficult for an attacker to identify that any given traffic is a part of the network.

To help prevent network failure, an anonymity network must avoid reliance on centralised servers. The practical problem with any peer-to-peer network is any client that wishes to connect to the network needs to know about at least one node before it's able to join. Known centralised points in any network are prone to attack.

Since the leaking of documents by Edward Snowden, online privacy has moved to the forefront of the minds of many internet users. This research will be of interest to privacy advocates who see the benefit of allowing personal communication online away from prying eyes. Also the technical challenges of this project will answer questions about how we keep our private data private.

A.3 Proposed Work

Development of a product for this project is dependent upon three key pieces of research.

An analysis will need to be conducted of algorithms of a Distributed Hash Table; this analysis will form the basis of my literature review. The comparison of two common DHT methods, Chord and Kademlia, will be the starting point for the analysis. The analysis will involve comparisons of different DHT algorithms based on three criteria:

- Speed—How quickly one node can find another node in the network.
- Robustness—How well the algorithm manages failures in the network.
- Security—How well the algorithm manages malicious nodes in the network.

The analysis will inform the decision of which DHT algorithm to use, after which the product will be designed by producing class diagrams and communication diagrams. This will form the basis for the implementation.

An analysis will need to be conducted on protocols for relaying information between nodes on the system. This will involve identifying an appropriate, efficient and stable protocol to allow nodes to securely communicate with one another.

A simple messaging protocol will be developed on-top of the P2P network to allow nodes to send basic text messages to one another. A user should be able to type a message and send it to another user on the network, that user should be able to receive the message and have it displayed instantly.

Research will be conducted to identify security and anonymity features which can be implemented in the system. The technique of onion routing has already been identified, and it will be implemented.

The product will be developed in Python.

A.4 Aims of Project

1. To investigate how peer-to-peer anonymity networks protect the information shared by its users.
2. To build a prototype instant-messaging application in an anonymity network, allowing users to securely share messages.

A.5 Objectives

- Research implementations of peer-to-peer networks.
- Research techniques for maintaining anonymity.
- Analyse DHT algorithms and to determine an appropriate one for the network.
- Create design documentation, defining the protocol for sharing messages.
- Implement a DHT algorithm to construct a peer-to-peer network.
- Implement a relay protocol to secure connections between nodes.
- Implement anonymity features.
- Successfully connect nodes together in a network.
- Demonstrate a working network by showing nodes connecting to one another and transferring messages.
- Evaluate the success of the network.

A.6 Skills

- **System Design**
  There will be a fair amount of code involved in the project, as such it's critical that the design of the product gives a good solid basis for implementation. The planning and design skills from acquired from the first year CM0432 Systems Analysis module and the second year CM0571 Professional Software Engineering Practice will be necessary to ensure that the system is designed correctly to avoid bugs appearing later on.
- **Computer Networks**
  A good basic knowledge of computer networks was gained from the EN0574 Computer Networks module that was completed in the second year, though more work on developing knowledge in this area will be necessary to be able to successfully implement more complex peer-to-peer based networks.
- **Python Programming**
  A successful placement completed in the Scientific Information Service at CERN gave plentiful experience of Python. Using additional libraries in Python will give more experience in this area.
- **Linux**
  The project will be completed in a Linux environment. Experience gained from the placement year and through several modules on the Computer Science course will form a good basis for working on Linux.

A.7 Sources of Information

A.8 Resources—statement of hardware & software required

- **Hardware**
  - Desktop Computer
  - Multiple Networked Computers (For Demonstration)
    Computers in Labs F1 or S2 can be used for this.
- **Software**
  - Linux OS (Ubuntu 14.04 LTS)
  - Oracle VirtualBox (For testing)
    It would be disruptive to take over multiple computers in F1 and S2, a virtual network and virtual nodes can be created with Oracle VirtualBox.
  - Text Editor (Sublime)
  - Python interpreter, version 2.7
  - PIP and VirtualEnv
  - Git (And GitHub)

A.9 Structure and Contents of project report

A.9.1 Report Structure

- Abstract

1. Introduction
2. Analysis
   (a) Distributed Hash Table—A comparison of DHT algorithms, how they work and their strengths and weaknesses.
   (b) Relaying—A comparison of relaying methods, with particular attention drawn to blocking resistance ability.
   (c) Anonymising—An investigation into anonymising techniques within peer-to-peer networks.
3. Synthesis
   (a) Design—System design, including class diagrams of the project, how the node software will be designed.

(b) Communication—Protocol design, including communications diagrams, how the network will be created and how nodes will share information.

(c) Implementation—Discussion of the challenges faced during implementation of the product.

(d) Testing—Testing plans which will demonstrate the working functionality of the product.

4. Evaluation
5. Conclusion
6. Bibliography
7. Appendices

A.9.2 List of Appendices

- Terms of Reference
- Requirements Specification
- Design Documentation
- Testing Results
- Source Code

A.10 Marking Scheme

A.10.1 Project Type

This is a Software Engineering Project.

Mark Allocation:

- Report (40%)
- Product (50%)
- Viva (10%)

A.10.2 Project Report

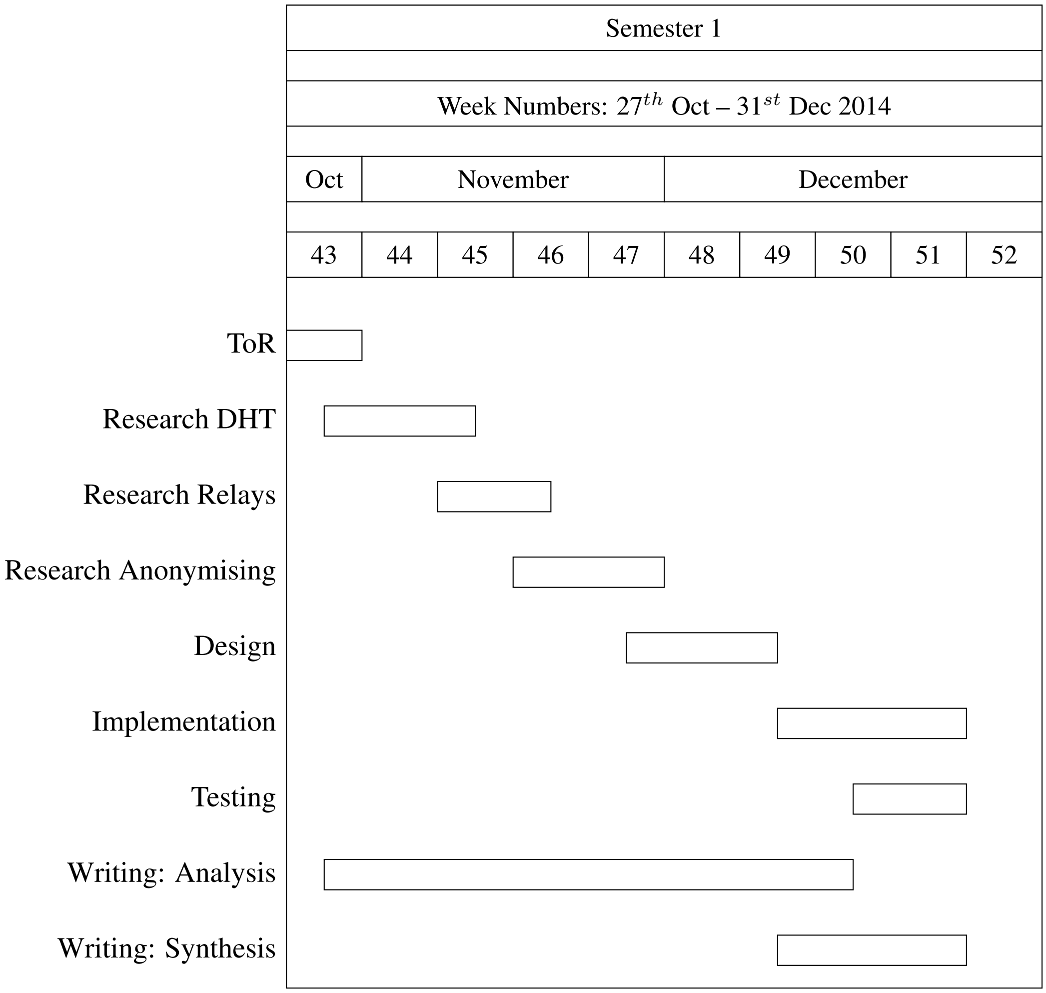
- Abstract and Introduction (5%)
- Analysis (30%)
  - Distributed Hash Table
  - Relaying

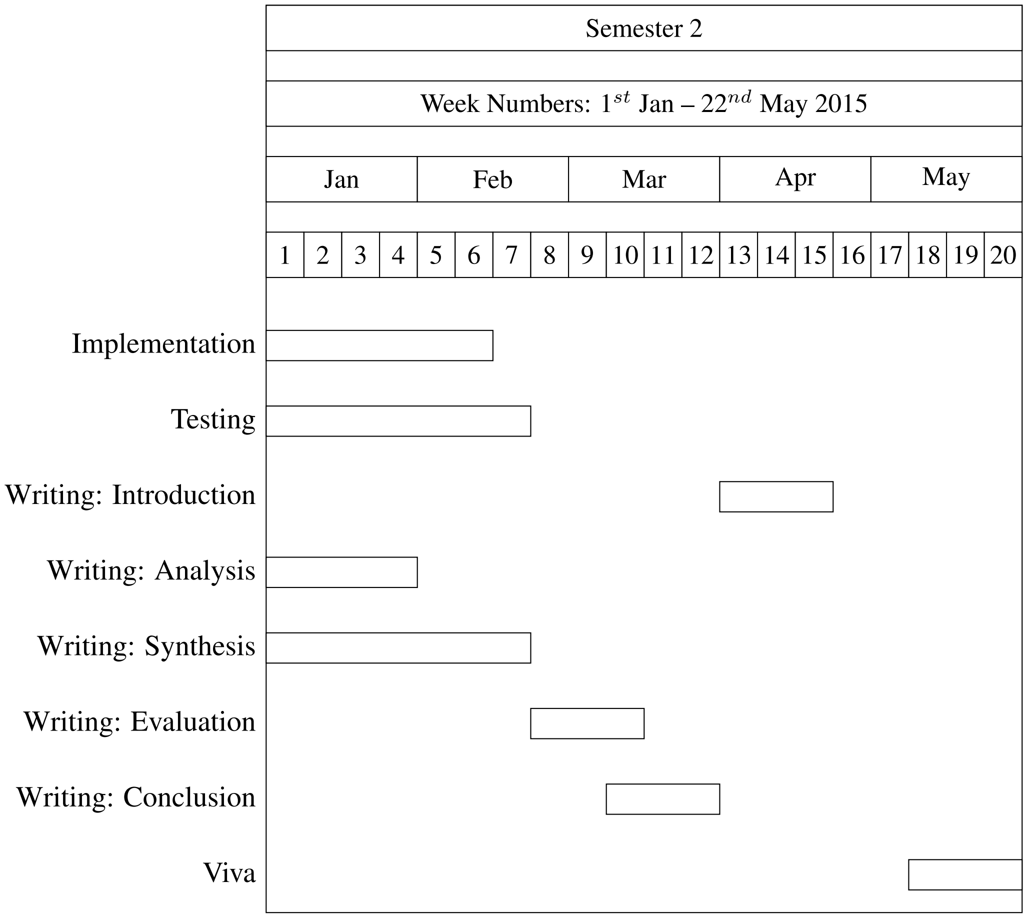
    - Anonymising
- Synthesis (30%)
    - Design
    - Communication
    - Implementation
    - Testing
- Evaluation and Conclusions (30%)

A.10.3 Product

- Fitness for Purpose (40%)
    - Completeness of prototype
    - Robustness of network
    - Successful sending of messages
- Build Quality (60%)
    - Design Documentation
    - Code Quality
    - Testing (Unit tests and successful demonstration)

A.11 Project Plan




Semester 2
Week Numbers: 1st Jan – 22nd May 2015
Jan
Feb
Mar
Apr
May
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Implementation
Testing
Writing: Introduction
Writing: Analysis
Writing: Synthesis
Writing: Evaluation
Writing: Conclusion
Viva

Ethical category of project *[Complete after approval]*	
Red	☐
Amber	☐
Green	☒

Department of Computer Science and Digital Technologies

UNDERGRADUATE PROJECT: ETHICS REGISTRATION AND APPROVAL FORM

Section One: Registration *[To be completed by student]*

Title of project	Secure messaging within a decentralised anonymity network.
Researcher's name	Graham Armstrong
Programme of study	Computer Science
Academic Year	4th Year
Module code	CM0645
Supervisor's name	David Kendall
Second Marker's name	Dave Harrison
Start date of project	08/10/14

Short description of project, including research methods and selection of any participants:

A Software Engineering project, investigating peer-to-peer anonymity networks and the application of a basic service.

Research will involve literature review of algorithm methodologies and testing of implemented code.

No participants are needed for the project.

Ethical considerations in the research project	**YES**	**NO**
1. Does your research involve an external organisation or partner e.g. NHS, School etc.		☒
2. Does your research involve human participants?		☒
3. If yes to Q.2, will you inform the participants about the research?		
4. Will you obtain their consent using the standard consent form?		
5. Is any deception involved?		
6. Do any participants constitute a 'vulnerable group' (see definition of Vulnerable People)		
7. Will the research involve the following information?		
Commercially sensitive	☐	☒
Personally sensitive	☐	☒
Politically sensitive	☐	☒
8. Is the research likely to cause any significant environmental impacts?	☐	☒
9. Are there likely to be any risks for you or for the participants in your research?	☐	☒
10. If yes [to 5, 6, 7, 8 or 9 above] have you identified steps to address the issues?		

Statement by researcher

This statement should explain how any issues identified in the answers to the above questions will be addressed and what steps will be taken to mitigate such risks or adverse impact

N/A

I have read the University and the Faculty Ethics Policy and Procedures and confirm that the answers I have given above are correct. Where issues arise under items 5, 6, 7, 8 or 9 [above] I have described in writing how I intend to approach these issues in the research.

Researcher's signature G. Armstrong

Date **24/10/2014**

Section Two: Approval

[The form is reviewed by the supervisor and second marker. Approval maybe given by either for green projects; amber projects must be approved by the second marker. Red projects must be referred to the Faculty Research Ethics Committee.]

***Red**: Vulnerable participants, sensitive data, risks to participants or researchers, NHS, etc.* ***Amber**: Human participants, environmental issues, commercially sensitive information, etc.* ***Green**: No participants involved, no sensitive data, etc.* *For full definitions see section on Risk Categories in the Engineering and Environment Ethics Procedures.*

Ethical approval
[Please tick as appropriate]

Green - Ethical approval is given without conditions	☒
Amber - Ethical approval is given with the following conditions • Information to be provided to all participants • Participant consent to be obtained using the standard Research Participant Consent Form or otherwise in accordance with Faculty procedures • Data to be stored and destroyed securely in accordance with University guidelines • Adherence to Data Protection Act • Anonymity to be provided to participants • Commercial confidentiality to be provided to organisations(s) • Other (please state):	☐ ☐ ☐ ☐ ☐ ☐ ☐
Red - Project is referred to FREC for approval	☐

Name of Approver	David Kendall
Signature	
Date	24/10/2014

Outcome of FREC referral – Decision, minute and date of meeting, or signatures of two signatories, one of whom is a member of FREC.

APPENDIX

B

REQUIREMENTS SPECIFICATION

This appendix lists the specific requirements of the project product. Commentary for these items is available in chapter 3.

B.1 Functional Requirements

B.1.1 High-Level Functional Requirements

These are the expected features of the product, they give a brief description of the behaviours expected of the final product.

1. It **must** be possible to start a node as though it were to form a new network, acting as an initial node.
2. It **must** be possible to start a node and make it join an existing network with the IP address and listening port of any existing node in the network.
3. The system **must** be able to successfully locate and communicate with any other node in the network. In this instance, communicate means any data transfer or request; not simply messaging.
4. The system **must** be able to send basic text messages directly between nodes.
5. The system **must** be able to send basic text messages between nodes that are relayed via multiple other nodes using an onion-style technique.

B.1.2 User Interface Requirements

The product...

6. **Must** provide a simple command prompt interface for which the user can input commands to control the node.
7. **Must** allow the user to send messages to a specified foreign node.
8. **Should** allow the user to request information about the finger of this node.

9. **Should** allow the user to request information about the fingers of other nodes.

B.1.3 Communication Requirements

The product...

10. **Must** encrypt outgoing communications and decrypt incoming ones.
11. **Must** respond promptly and appropriately to valid requests for information in the DHT.
12. **Must** perform basic verification to try and ensure the incoming connections are honest.
13. **Should** handle incoming connections without disrupting the user.
14. **Could** use random padding when transmitting messages.
15. **Could** send and receive finger information in messages, using this information to update the finger table.
16. **Would not** continue communication with a node deemed dishonest.

B.2 Non-Functional Requirements

B.2.1 Data Structure Requirements

The product...

17. **Must** have a structure to represent a Finger in the system; a Finger contains the IP address, listening port, and public key of a node.
18. **Must** have a structure for storing and controlling access to fingers, a finger table.
19. **Should** structure finger information in a tree-like manner, similar in design to Kademlia.

B.2.2 Network Requirements

The product...

20. **Must** generate unique node identifiers using a hashing algorithm.
21. **Should** work across Local Area Networks using IPv4.
22. **Could** provide heart-beat functionality to detect failed or dropped nodes.

B.2.3 Security Requirements

The product...

23. **Must** implement a form of asymmetric public-key cryptography.

24. **Must** implement a secure collision-resistant hashing algorithm.
25. **Should** include random padding when transmitting messages.
26. **Could** implement additional blocking resistance features.

B.2.4 Operating Requirements

The product...

27. **Must** be functional within a Linux environment.
28. **Should** be functional within the environment of any other Operating System for which a Python environment is possible.
29. **Should** be error-tolerant; errors should not cause the entire system to fail.

APPENDIX

C

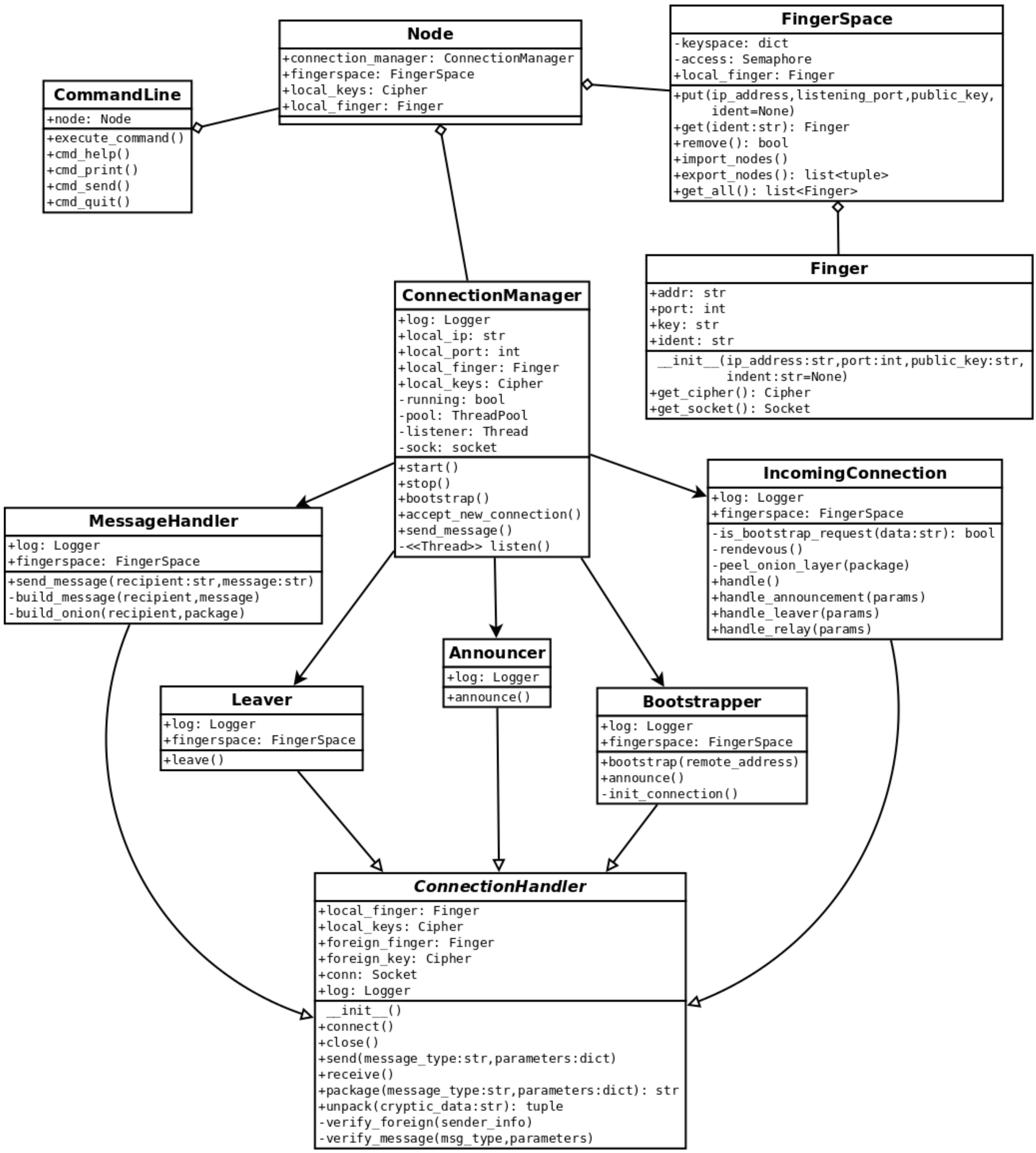
DESIGN DOCUMENTATION

This appendix contains the design deliverables of the product; this includes:

- UML Class Diagram, detailing the relationship of the main classes within the product.
- UML Sequence Diagrams, demonstrating use cases of the protocol which has been developed for this product.

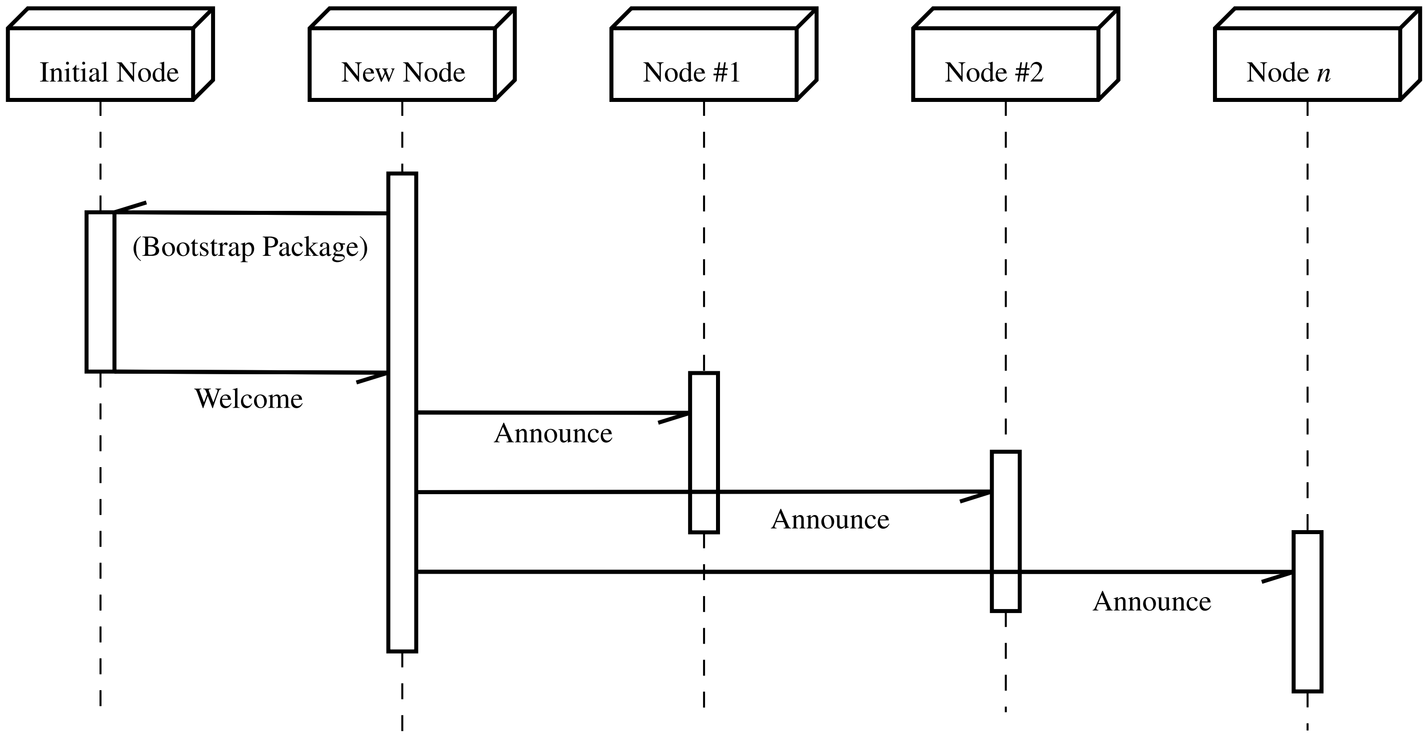
The design deliverables are discussed in more detail in chapter 4.

C.1 Class Diagram

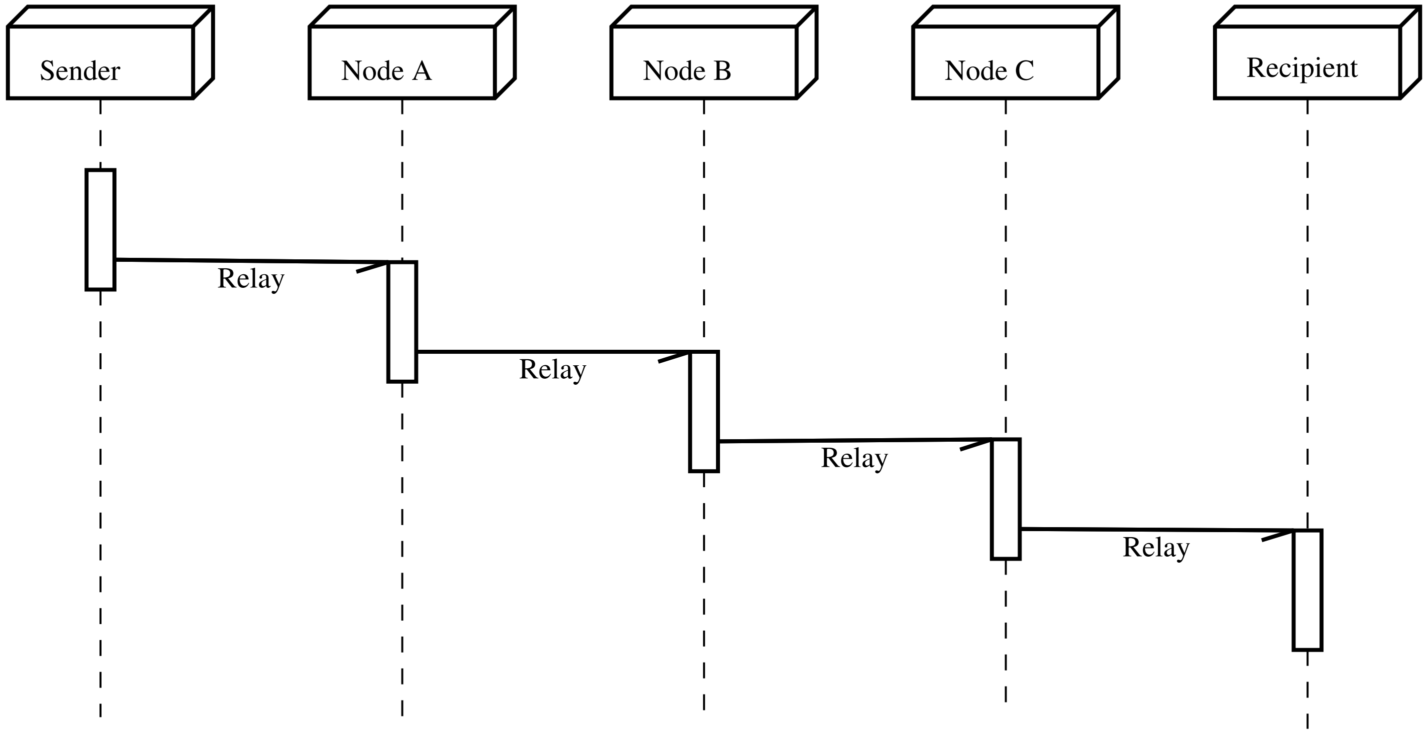


C.2 Sequence Diagrams

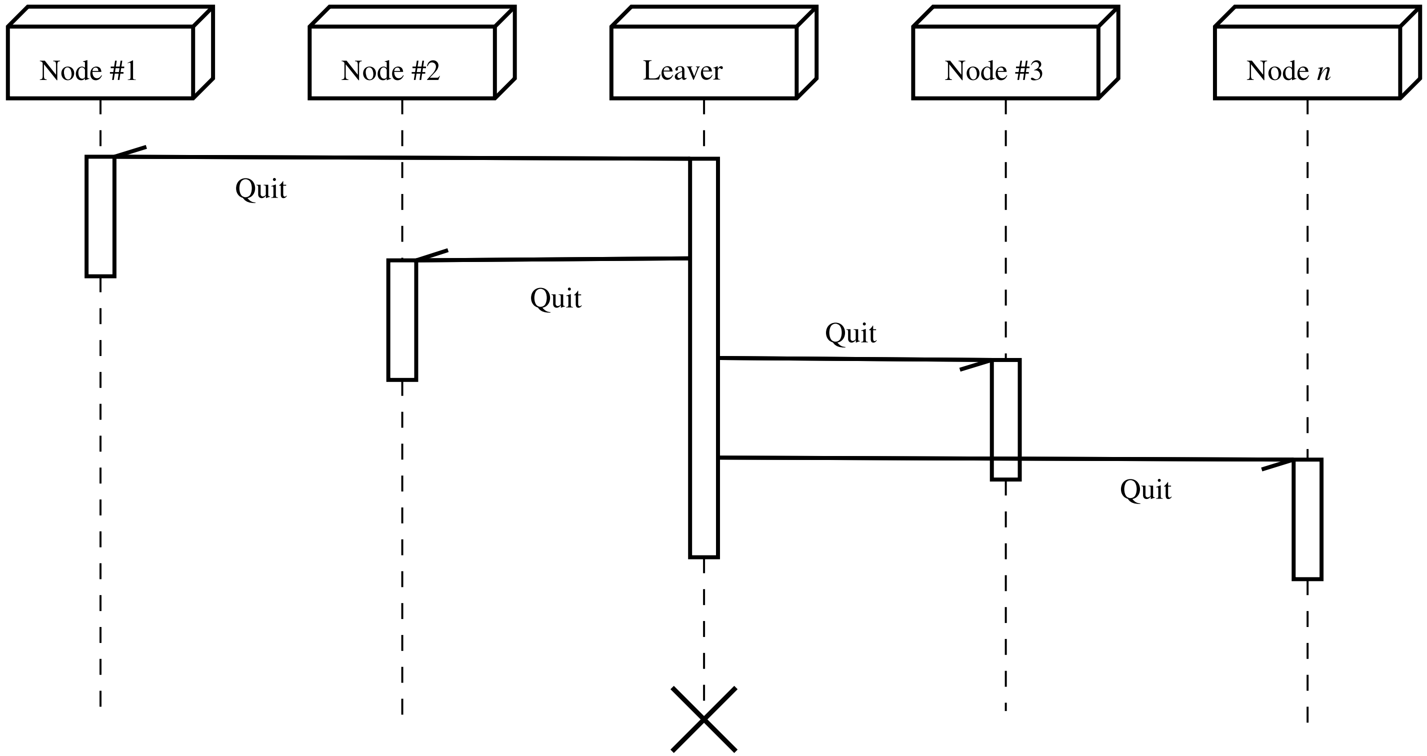
C.2.1 Node Joining



C.2.2 Message Relaying



C.2.3 Node Departure



APPENDIX

D

TESTING RESULTS

This appendix lists the results of the tests that were planned during synthesis, section 4.4. This document shows the results of unit tests and integration tests. *All unit tests and integration tests are passing!*

Note that all integration tests are executed from the root of the product directory.

Unit Tests

The unit tests are completed by executing PY.TEST within the product directory. Unit tests are found and executed. The results are below, 47 individual tests were executed and found to pass.

Listing D.1: Unit Test Results

```
$ py.test
============================ test session starts =============================
platform linux2 -- Python 2.7.6 -- py-1.4.26 -- pytest-2.6.4
collected 47 items

code/distrim/unit_tests/test_fingerspace.py ..................
code/distrim/unit_tests/test_protocol.py .......
code/distrim/utils/unit_tests/test_utilities.py ......................

========================= 47 passed in 23.11 seconds =========================
```

D.1 Initialisation Test

Demonstration of node initialisation.

Listing D.2: Initial Node

```
$ ./run
[13:07:37] [INFO] <MainThread>: Node started 28ff @ 192.168.0.2:2000
[13:07:37] [INFO] <MainThread>: Listening for connections on 192.168.0.2:2000
> help
DistrIM Commands
  help,   h:  Print this message.
  print,  p:  Print some information.
```

```
  send,   s:  Send message to node.
  quit,   q:  Stop this node and exit.
> print
Possible options:
  crypto-keys
  fingers
  node-info
  node-stats
```

The node is started successfully and begins listening on the default port 2000, the arrow symbol is the prompt for user input. The HELP command prints to screen the commands available to the user, this is shown here for demonstration purposes.

It can be shown that the port is now listening for connections by using the NETSTAT command.

Listing D.3: Netstat Response, the port is open.

```
$ netstat -vatn | grep 2000
tcp        0      0 192.168.0.2:2000        0.0.0.0:*               LISTEN
```

D.2 Bootstrap Test

Demonstration of the bootstrapping procedure. The new node connects to the initial node, the finger tables confirm they now know about one another.

Listing D.4: Initial Node 018c

```
$ ./run
[13:29:04] [INFO] <MainThread>: Node started 018c @ 192.168.0.2:2000
[13:29:04] [INFO] <MainThread>: Listening for connections on 192.168.0.2:2000
> [13:29:07] [INFO] <Thread-1>: New Connection from: ('192.168.0.2', 58165)
[13:29:07] [INFO] <Thread-1>: New node joining network with ID: 4550
[13:29:07] [INFO] <Thread-1>: Sending welcome message to 4550
print fingers
Finger Table...
 4550) 192.168.0.2:2001
>
```

Listing D.5: New Node 4550

```
$ ./run -b 192.168.0.2:2000 -p 2001
[13:29:07] [INFO] <MainThread>: Node started 4550 @ 192.168.0.2:2001
[13:29:07] [INFO] <MainThread>: Listening for connections on 192.168.0.2:2001
[13:29:07] [INFO] <MainThread>: Boostrapping to 192.168.0.2:2000
[13:29:07] [DEBUG] <MainThread>: Bootstrap connection established.
[13:29:07] [DEBUG] <MainThread>: Bootstrap package sent.
[13:29:07] [INFO] <MainThread>: SUCCESS! Rendezvous occured.
> print fingers
Finger Table...
 018c) 192.168.0.2:2000
>
```

D.3 Announcing and Rendezvous Test

Demonstration of network distribution. When the two new nodes connect to the initial node, they become aware of each other because Node B has received an announcement from node C. All nodes then poses the fingers of the other two nodes.

Listing D.6: Initial Node A

```
$ ./run
[13:39:53] [INFO] <MainThread>: Node started 3290 @ 192.168.0.2:2000
[13:39:53] [INFO] <MainThread>: Listening for connections on 192.168.0.2:2000
> [13:40:02] [INFO] <Thread-1>: New Connection from: ('192.168.0.2', 58177)
[13:40:02] [INFO] <Thread-1>: New node joining network with ID: f0ec
[13:40:02] [INFO] <Thread-1>: Sending welcome message to f0ec
[13:40:06] [INFO] <Thread-2>: New Connection from: ('192.168.0.2', 58179)
[13:40:06] [INFO] <Thread-2>: New node joining network with ID: 0327
[13:40:06] [INFO] <Thread-2>: Sending welcome message to 0327
print fingers
Finger Table...
 f0ec) 192.168.0.2:2001
 0327) 192.168.0.2:2002
>
```

Listing D.7: New Node B

```
$ ./run -b 192.168.0.2:2000 -p 2001
[13:40:02] [INFO] <MainThread>: Node started f0ec @ 192.168.0.2:2001
[13:40:02] [INFO] <MainThread>: Listening for connections on 192.168.0.2:2001
[13:40:02] [INFO] <MainThread>: Boostrapping to 192.168.0.2:2000
[13:40:02] [DEBUG] <MainThread>: Bootstrap connection established.
[13:40:02] [DEBUG] <MainThread>: Bootstrap package sent.
[13:40:02] [INFO] <MainThread>: SUCCESS! Rendezvous occured.
> [13:40:06] [INFO] <Thread-1>: New Connection from: ('192.168.0.2', 56686)
[13:40:06] [DEBUG] <Thread-1>: Authenticating new connection...
[13:40:06] [INFO] <Thread-1>: Announcement from 0327
print fingers
Finger Table...
 3290) 192.168.0.2:2000
 0327) 192.168.0.2:2002
>
```

Listing D.8: New Node C

```
$ ./run -b 192.168.0.2:2000 -p 2002
[13:40:06] [INFO] <MainThread>: Node started 0327 @ 192.168.0.2:2002
[13:40:06] [INFO] <MainThread>: Listening for connections on 192.168.0.2:2002
[13:40:06] [INFO] <MainThread>: Boostrapping to 192.168.0.2:2000
[13:40:06] [DEBUG] <MainThread>: Bootstrap connection established.
[13:40:06] [DEBUG] <MainThread>: Bootstrap package sent.
[13:40:06] [INFO] <MainThread>: Announce to <Fingerspace.Finger f0ec @
    192.168.0.2:2001>
[13:40:06] [INFO] <MainThread>: SUCCESS! Rendezvous occured.
> print fingers
Finger Table...
 3290) 192.168.0.2:2000
 f0ec) 192.168.0.2:2001
>
```

D.4 Departure Integrity Test

Demonstration of network distribution. When the two new nodes connect to the initial node, they become aware of each other because Node B receives an announcement from node C. All nodes poses the fingers of the other two nodes.

When node A departs, nodes B and C still have each other in their finger table; but A has been removed.

Listing D.9: Initial Node A

```
$ ./run
[13:45:27] [INFO] <MainThread>: Node started 7a44 @ 192.168.0.2:2000
[13:45:27] [INFO] <MainThread>: Listening for connections on 192.168.0.2:2000
> [13:45:29] [INFO] <Thread-1>: New Connection from: ('192.168.0.2', 58186)
[13:45:29] [INFO] <Thread-1>: New node joining network with ID: 3157
[13:45:29] [INFO] <Thread-1>: Sending welcome message to 3157
[13:45:31] [INFO] <Thread-2>: New Connection from: ('192.168.0.2', 58187)
[13:45:31] [INFO] <Thread-2>: New node joining network with ID: 550f
[13:45:31] [INFO] <Thread-2>: Sending welcome message to 550f
print fingers
Finger Table...
 550f) 192.168.0.2:2002
 3157) 192.168.0.2:2001
> quit
Shutting down...
[13:45:54] [INFO] <MainThread>: Node Stopping...
[13:45:54] [DEBUG] <Thread-Listener>: Listening thread stopped.
```

Listing D.10: Node B

```
$ ./run -b 192.168.0.2:2000 -p 2001
[13:45:29] [INFO] <MainThread>: Node started 3157 @ 192.168.0.2:2001
[13:45:29] [INFO] <MainThread>: Listening for connections on 192.168.0.2:2001
[13:45:29] [INFO] <MainThread>: Boostrapping to 192.168.0.2:2000
[13:45:29] [DEBUG] <MainThread>: Bootstrap connection established.
[13:45:29] [DEBUG] <MainThread>: Bootstrap package sent.
[13:45:29] [INFO] <MainThread>: SUCCESS! Rendezvous occured.
> [13:45:31] [INFO] <Thread-1>: New Connection from: ('192.168.0.2', 56694)
[13:45:31] [DEBUG] <Thread-1>: Authenticating new connection...
[13:45:31] [INFO] <Thread-1>: Announcement from 550f
print fingers
Finger Table...
 7a44) 192.168.0.2:2000
 550f) 192.168.0.2:2002
> [13:45:54] [INFO] <Thread-2>: New Connection from: ('192.168.0.2', 56696)
[13:45:54] [DEBUG] <Thread-2>: Authenticating new connection...
[13:45:54] [INFO] <Thread-2>: Goodbye to 7a44
print fingers
Finger Table...
 550f) 192.168.0.2:2002
>
```

Listing D.11: Node C

```
$ ./run -b 192.168.0.2:2000 -p 2002
[13:45:31] [INFO] <MainThread>: Node started 550f @ 192.168.0.2:2002
```

```
[13:45:31] [INFO] <MainThread>: Listening for connections on 192.168.0.2:2002
[13:45:31] [INFO] <MainThread>: Boostrapping to 192.168.0.2:2000
[13:45:31] [DEBUG] <MainThread>: Bootstrap connection established.
[13:45:31] [DEBUG] <MainThread>: Bootstrap package sent.
[13:45:31] [INFO] <MainThread>: Announce to <Fingerspace.Finger 3157 @
    192.168.0.2:2001>
[13:45:31] [INFO] <MainThread>: SUCCESS! Rendezvous occured.
> print fingers
Finger Table...
 7a44) 192.168.0.2:2000
 3157) 192.168.0.2:2001
> [13:45:54] [INFO] <Thread-1>: New Connection from: ('192.168.0.2', 39417)
[13:45:54] [DEBUG] <Thread-1>: Authenticating new connection...
[13:45:54] [INFO] <Thread-1>: Goodbye to 7a44
print fingers
Finger Table...
 3157) 192.168.0.2:2001
>
```

D.5 Single Message Passing Test

This test shows that two nodes can pass messages between themselves. When the two nodes rendezvous, they use the SEND command to communicate with each other.

Listing D.12: Node A

```
$ ./run
[14:01:30] [INFO] <MainThread>: Node started 94a4 @ 192.168.0.2:2000
[14:01:30] [INFO] <MainThread>: Listening for connections on 192.168.0.2:2000
> [14:02:10] [INFO] <Thread-1>: New Connection from: ('192.168.0.2', 58231)
[14:02:10] [INFO] <Thread-1>: New node joining network with ID: ecdc
[14:02:10] [INFO] <Thread-1>: Sending welcome message to ecdc
> send ecdc Hello there! Isn't it a beautiful day?
[14:03:04] [DEBUG] <MainThread>: Path Length 0
[14:03:04] [DEBUG] <MainThread>: Path: ecdc <-
> [14:03:30] [INFO] <Thread-2>: New Connection from: ('192.168.0.2', 58235)
[14:03:31] [DEBUG] <Thread-2>: Authenticating new connection...
[14:03:31] [INFO] <Thread-2>: Message Received from ecdc
## Message: Why yes, it's positively delightful!
```

Listing D.13: Node B

```
$ ./run -b 192.168.0.2:2000 -p 2001
[14:02:10] [INFO] <MainThread>: Node started ecdc @ 192.168.0.2:2001
[14:02:10] [INFO] <MainThread>: Listening for connections on 192.168.0.2:2001
[14:02:10] [INFO] <MainThread>: Boostrapping to 192.168.0.2:2000
[14:02:10] [DEBUG] <MainThread>: Bootstrap connection established.
[14:02:10] [DEBUG] <MainThread>: Bootstrap package sent.
[14:02:10] [INFO] <MainThread>: SUCCESS! Rendezvous occured.
> [14:03:04] [INFO] <Thread-1>: New Connection from: ('192.168.0.2', 56740)
[14:03:04] [DEBUG] <Thread-1>: Authenticating new connection...
[14:03:04] [INFO] <Thread-1>: Message Received from 94a4
## Message: Hello there! Isn't it a beautiful day?
send 94a4 Why yes, it's positively delightful!
[14:03:30] [DEBUG] <MainThread>: Path Length 0
```

```
[14:03:30] [DEBUG] <MainThread>: Path: 94a4 <-
>
```

D.6 Onion Routing Test

This test demonstrates the working functionality of message passing via onion routing. Two nodes A and B are created, followed by five additional nodes; node A will send a message to node B. The finger tables of nodes A and B show all other nodes on the network. Once initial connections are established and discovery has taken place, the message is sent. The remote logger shows the message relaying between nodes. Node B then receives the message.

Note, a *remote logger* was implemented as part of the project; it's shown in use here, the five additional nodes output their messages to it.

Listing D.14: Node A

```
$ ./run
[14:13:52] [INFO] <MainThread>: Node started 66a4 @ 192.168.0.2:2000
[14:13:52] [INFO] <MainThread>: Listening for connections on 192.168.0.2:2000
> [14:14:11] [INFO] <Thread-1>: New Connection from: ('192.168.0.2', 58372)
[14:14:11] [INFO] <Thread-1>: New node joining network with ID: 370f
[14:14:11] [INFO] <Thread-1>: Sending welcome message to 370f
[14:14:42] [INFO] <Thread-2>: New Connection from: ('192.168.0.2', 58373)
[14:14:42] [INFO] <Thread-2>: New node joining network with ID: ac2a
[14:14:42] [INFO] <Thread-2>: Sending welcome message to ac2a
[14:15:06] [INFO] <Thread-3>: New Connection from: ('192.168.0.2', 58376)
[14:15:06] [DEBUG] <Thread-3>: Authenticating new connection...
[14:15:06] [INFO] <Thread-3>: Announcement from 649d
[14:15:20] [INFO] <Thread-4>: New Connection from: ('192.168.0.2', 58380)
[14:15:20] [DEBUG] <Thread-4>: Authenticating new connection...
[14:15:20] [INFO] <Thread-4>: Announcement from db59
[14:15:38] [INFO] <Thread-5>: New Connection from: ('192.168.0.2', 58383)
[14:15:38] [INFO] <Thread-5>: New node joining network with ID: b456
[14:15:38] [INFO] <Thread-5>: Sending welcome message to b456
[14:15:52] [INFO] <Thread-6>: New Connection from: ('192.168.0.2', 58389)
[14:15:52] [DEBUG] <Thread-6>: Authenticating new connection...
[14:15:52] [INFO] <Thread-6>: Announcement from 1bd9
print fingers
Finger Table...
 ac2a) 192.168.0.2:2002
 370f) 192.168.0.2:2001
 b456) 192.168.0.2:2005
 1bd9) 192.168.0.2:2006
 db59) 192.168.0.2:2004
 649d) 192.168.0.2:2003
> send 370f This message will be layered, like a delicious onion!
[14:17:07] [DEBUG] <MainThread>: Path Length 4
[14:17:07] [DEBUG] <MainThread>: Path: 370f <- 649d <-- ac2a <-- db59 <-- 1bd9
>
```

Listing D.15: Node B

```
$ ./run -b 192.168.0.2:2000 -p 2001
[14:14:11] [INFO] <MainThread>: Node started 370f @ 192.168.0.2:2001
```

```
[14:14:11] [INFO] <MainThread>: Listening for connections on 192.168.0.2:2001
[14:14:11] [INFO] <MainThread>: Boostrapping to 192.168.0.2:2000
[14:14:11] [DEBUG] <MainThread>: Bootstrap connection established.
[14:14:11] [DEBUG] <MainThread>: Bootstrap package sent.
[14:14:11] [INFO] <MainThread>: SUCCESS! Rendezvous occured.
> [14:14:42] [INFO] <Thread-1>: New Connection from: ('192.168.0.2', 56880)
[14:14:42] [DEBUG] <Thread-1>: Authenticating new connection...
[14:14:42] [INFO] <Thread-1>: Announcement from ac2a
[14:15:06] [INFO] <Thread-2>: New Connection from: ('192.168.0.2', 56883)
[14:15:06] [DEBUG] <Thread-2>: Authenticating new connection...
[14:15:06] [INFO] <Thread-2>: Announcement from 649d
[14:15:20] [INFO] <Thread-3>: New Connection from: ('192.168.0.2', 56884)
[14:15:20] [INFO] <Thread-3>: New node joining network with ID: db59
[14:15:20] [INFO] <Thread-3>: Sending welcome message to db59
[14:15:38] [INFO] <Thread-4>: New Connection from: ('192.168.0.2', 56893)
[14:15:38] [DEBUG] <Thread-4>: Authenticating new connection...
[14:15:38] [INFO] <Thread-4>: Announcement from b456
[14:15:52] [INFO] <Thread-5>: New Connection from: ('192.168.0.2', 56897)
[14:15:52] [DEBUG] <Thread-5>: Authenticating new connection...
[14:15:52] [INFO] <Thread-5>: Announcement from 1bd9
print fingers
Finger Table...
 66a4) 192.168.0.2:2000
 ac2a) 192.168.0.2:2002
 b456) 192.168.0.2:2005
 1bd9) 192.168.0.2:2006
 db59) 192.168.0.2:2004
 649d) 192.168.0.2:2003
> print node-info
Node Information...
    Hash: 370f
 Node IP: 192.168.0.2:2001
 Pub-Key: -----BEGIN PUBLIC KEY-----
          MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCise/sF6yHDYtMXJOJk6AJMixN
          lClMFTl6zz/9UtNi9HbIyppEiJT9rymx0Iloovai2C49SQvFsHpYrhbR7d8OYDd/
          uLrmDf3WxjVn/BhboJR9rFxT0f772ukgibI848+J47cRB8kUsD1Nn5vgcorLPagk
          kx1wCL6CPjBSimlPfwIDAQAB
          -----END PUBLIC KEY-----
> [14:17:24] [INFO] <Thread-6>: New Connection from: ('192.168.0.2', 56904)
[14:17:24] [DEBUG] <Thread-6>: Authenticating new connection...
[14:17:24] [INFO] <Thread-6>: Message Received from 66a4
## Message: This message will be layered, like a delicious onion!
```

Listing D.16: Logger

```
$ ./listen
Listening on 127.0.0.1:1999 ...
 ac2a | [14:14:42] [INFO] <MainThread>: Node started ac2a @ 192.168.0.2:2002
 ac2a | [14:14:42] [INFO] <MainThread>: Listening for connections on
    192.168.0.2:2002
 ac2a | [14:14:42] [INFO] <MainThread>: Boostrapping to 192.168.0.2:2000
 ac2a | [14:14:42] [INFO] <MainThread>: Announce to <Fingerspace.Finger 370f @
    192.168.0.2:2001>
 ac2a | [14:14:42] [INFO] <MainThread>: SUCCESS! Rendezvous occured.
 649d | [14:15:06] [INFO] <MainThread>: Node started 649d @ 192.168.0.2:2003
 649d | [14:15:06] [INFO] <MainThread>: Listening for connections on
    192.168.0.2:2003
 649d | [14:15:06] [INFO] <MainThread>: Boostrapping to 192.168.0.2:2002
```

```
ac2a | [14:15:06] [INFO] <Thread-1>: New Connection from: ('192.168.0.2',
   39603)
ac2a | [14:15:06] [INFO] <Thread-1>: New node joining network with ID: 649d
ac2a | [14:15:06] [INFO] <Thread-1>: Sending welcome message to 649d
649d | [14:15:06] [INFO] <MainThread>: Announce to <Fingerspace.Finger 66a4 @
   192.168.0.2:2000>
649d | [14:15:06] [INFO] <MainThread>: Announce to <Fingerspace.Finger 370f @
   192.168.0.2:2001>
649d | [14:15:06] [INFO] <MainThread>: SUCCESS! Rendezvous occured.
db59 | [14:15:20] [INFO] <MainThread>: Node started db59 @ 192.168.0.2:2004
db59 | [14:15:20] [INFO] <MainThread>: Listening for connections on
   192.168.0.2:2004
db59 | [14:15:20] [INFO] <MainThread>: Boostrapping to 192.168.0.2:2001
db59 | [14:15:20] [INFO] <MainThread>: Announce to <Fingerspace.Finger ac2a @
   192.168.0.2:2002>
ac2a | [14:15:20] [INFO] <Thread-2>: New Connection from: ('192.168.0.2',
   39607)
db59 | [14:15:20] [INFO] <MainThread>: Announce to <Fingerspace.Finger 66a4 @
   192.168.0.2:2000>
db59 | [14:15:20] [INFO] <MainThread>: Announce to <Fingerspace.Finger 649d @
   192.168.0.2:2003>
649d | [14:15:20] [INFO] <Thread-1>: New Connection from: ('192.168.0.2',
   33806)
db59 | [14:15:20] [INFO] <MainThread>: SUCCESS! Rendezvous occured.
ac2a | [14:15:20] [INFO] <Thread-2>: Announcement from db59
649d | [14:15:20] [INFO] <Thread-1>: Announcement from db59
b456 | [14:15:38] [INFO] <MainThread>: Node started b456 @ 192.168.0.2:2005
b456 | [14:15:38] [INFO] <MainThread>: Listening for connections on
   192.168.0.2:2005
b456 | [14:15:38] [INFO] <MainThread>: Boostrapping to 192.168.0.2:2000
b456 | [14:15:38] [INFO] <MainThread>: Announce to <Fingerspace.Finger db59 @
   192.168.0.2:2004>
db59 | [14:15:38] [INFO] <Thread-1>: New Connection from: ('192.168.0.2',
   49167)
b456 | [14:15:38] [INFO] <MainThread>: Announce to <Fingerspace.Finger ac2a @
   192.168.0.2:2002>
ac2a | [14:15:38] [INFO] <Thread-3>: New Connection from: ('192.168.0.2',
   39613)
b456 | [14:15:38] [INFO] <MainThread>: Announce to <Fingerspace.Finger 649d @
   192.168.0.2:2003>
649d | [14:15:38] [INFO] <Thread-2>: New Connection from: ('192.168.0.2',
   33811)
b456 | [14:15:38] [INFO] <MainThread>: Announce to <Fingerspace.Finger 370f @
   192.168.0.2:2001>
b456 | [14:15:38] [INFO] <MainThread>: SUCCESS! Rendezvous occured.
db59 | [14:15:38] [INFO] <Thread-1>: Announcement from b456
ac2a | [14:15:38] [INFO] <Thread-3>: Announcement from b456
649d | [14:15:38] [INFO] <Thread-2>: Announcement from b456
1bd9 | [14:15:52] [INFO] <MainThread>: Node started 1bd9 @ 192.168.0.2:2006
1bd9 | [14:15:52] [INFO] <MainThread>: Listening for connections on
   192.168.0.2:2006
1bd9 | [14:15:52] [INFO] <MainThread>: Boostrapping to 192.168.0.2:2005
b456 | [14:15:52] [INFO] <Thread-1>: New Connection from: ('192.168.0.2',
   60689)
b456 | [14:15:52] [INFO] <Thread-1>: New node joining network with ID: 1bd9
b456 | [14:15:52] [INFO] <Thread-1>: Sending welcome message to 1bd9
1bd9 | [14:15:52] [INFO] <MainThread>: Announce to <Fingerspace.Finger 66a4 @
```

```
   192.168.0.2:2000>
1bd9 | [14:15:52] [INFO] <MainThread>: Announce to <Fingerspace.Finger ac2a @
   192.168.0.2:2002>
ac2a | [14:15:52] [INFO] <Thread-4>: New Connection from: ('192.168.0.2',
   39618)
1bd9 | [14:15:52] [INFO] <MainThread>: Announce to <Fingerspace.Finger 370f @
   192.168.0.2:2001>
1bd9 | [14:15:52] [INFO] <MainThread>: Announce to <Fingerspace.Finger db59 @
   192.168.0.2:2004>
db59 | [14:15:52] [INFO] <Thread-2>: New Connection from: ('192.168.0.2',
   49175)
1bd9 | [14:15:52] [INFO] <MainThread>: Announce to <Fingerspace.Finger 649d @
   192.168.0.2:2003>
649d | [14:15:52] [INFO] <Thread-3>: New Connection from: ('192.168.0.2',
   33818)
1bd9 | [14:15:52] [INFO] <MainThread>: SUCCESS! Rendezvous occured.
ac2a | [14:15:52] [INFO] <Thread-4>: Announcement from 1bd9
db59 | [14:15:52] [INFO] <Thread-2>: Announcement from 1bd9
649d | [14:15:52] [INFO] <Thread-3>: Announcement from 1bd9
1bd9 | [14:17:07] [INFO] <Thread-1>: New Connection from: ('192.168.0.2',
   36198)
1bd9 | [14:17:19] [INFO] <Thread-1>: Relaying message from 66a4 to db59
db59 | [14:17:19] [INFO] <Thread-3>: New Connection from: ('192.168.0.2',
   49178)
db59 | [14:17:22] [INFO] <Thread-3>: Relaying message from 1bd9 to ac2a
ac2a | [14:17:22] [INFO] <Thread-5>: New Connection from: ('192.168.0.2',
   39624)
ac2a | [14:17:23] [INFO] <Thread-5>: Relaying message from db59 to 649d
649d | [14:17:23] [INFO] <Thread-4>: New Connection from: ('192.168.0.2',
   33822)
649d | [14:17:24] [INFO] <Thread-4>: Relaying message from ac2a to 370f
```

APPENDIX

E

PROJECT DOCUMENTATION

E.1 About

DistrIM is a DHT-based network for secured messaging.

- Author: Graham Armstrong
- Student ID: 11004764
- Email: <graham.armstrong@northumbria.ac.uk>

Product for CM0645 Individual Project, academic year 2014/15.

This product has been developed in partial fulfilment of the regulations governing the award of the Degree of BSc (Honours) Computer Science at the University of Northumbria at Newcastle.

E.2 Technical

E.2.1 Requirements

The necessary Python packages are defined in requirements.txt

- Python 2.7
- Pip (Python Package Index)

E.2.2 Setup

It's best to use a virtual environment.

Install the necessary Python packages as such:

```
$ pip install -r requirements.txt
```

E.2.3 Usage

Begin execution with the following command:

$ ./run

For assistance with parameters, try using the **-h** flag.

$ ./run -h

E.3 Main Modules

E.3.1 Connections

Connections deals with establishing connections to foreign nodes and listening out for incoming connections.

class `distrim.connections.`**`ConnectionsManager`**(*parent_log*, *local_ip*, *local_port*, *fingerspace*, *finger*, *keys*)

The ConnectionsManager class is responsible for all incoming connections from other nodes.

`__init__`(*parent_log*, *local_ip*, *local_port*, *fingerspace*, *finger*, *keys*)

Parameters **parent_log** –

`_cleaning`()

Removes completed connection results from the thread pool.

This method is the target of *self._cleaner*

`_listen`()

Listen for incoming connections.

This method is the target of *self._thread*

`accept_new_connetion`(*sock*, *address*)

Handle incoming connections

Parameters

- **sock** – The socket of the incoming connection.
- **address** – Address of the connecting node.

`bootstrap`(*remote_ip*, *remote_port*)

Establish the first connection in the network.

`pool_new_connection`(*sock*, *address*)

Handle incoming connection, puts socket into seperate thread.

`send_message` (*recipient*, *message*)
Send a message via relays.

Parameters

- **recipient** – Finger of the node to send the message to.
- **message** – Plaintext message to send.

`start` ()
Begin the listening and cleaning threads of this node.

`stop` ()
Stop listening for connections and end the listening thread.

E.3.2 Fingerspace

Fingerspace Documentation

class `distrim.fingerspace.Finger` (*ip_address*, *listening_port*, *public_key*, *ident*='')
Contains identifying information unique to a single node.

This class represents the information identifying a particular Node in the network. Provides some functionality for connecting and communicating with the node.

Four attributes are stored:

- The ident of the node.
- The IP address of the node.
- The listening port of node.
- The public key of the node.

These are the attributes needed for communication between nodes.

On instantiation, an optional identifier value can also be passed in; the ident is calculated anyway but if given then it can be validated for authenticity.

`__eq__` (*other*)
Check if this finger is equal to another.

Parameters **other** – The other finger.

`__init__` (*ip_address*, *listening_port*, *public_key*, *ident*='')

Parameters

- **ip_address** – IP address of the node.
- **listening_port** – Listening port of the node.
- **public_key** – Public key of the node.
- **ident** – Hash value representing the identity of the node.

`__repr__`()
Representation of this object by text.

`get_cipher`()
Get an RSA cipher for message encryption.

Returns an instance of an RSA cipher of the Public Key of this Node.

Returns RSA Public Key instance of type `CipherWrap`.

`get_socket`()
Get a socket object connected to this node.

Returns A `SocketWrapper` instance with internal address defined, but not connected.

class `distrim.fingerspace.`**`FingerSpace`**(*parent_log*, *local_finger*)
The FingerSpace class is responsible for storing information about nodes. Access to the Key Space is managed through this class.

`__init__`(*parent_log*, *local_finger*)

Parameters

- **parent_log** – logger object from Node instance.
- **local_finger** – The finger for this node.

`__len__`()
FingerSpace length, the number of keys stored

`export_nodes`()
Export a list of all nodes.

Exports a list of all nodes in tuple format for serialising and sending to foreign nodes.

Data is exported as (ip address, port, public key, ident)

Returns A list of tuples with the data from the ‘all’ attribute.

`get`(*ident*)
Retrieve a Node Finger with the ident.

Parameters **ident** – ident of the Finger to fetch.

Returns Finger of the node, or *None* if node not found.

`get_all`()
Gets a list of all fingers.

`get_random_fingers`(*number*)
Get random fingers.

Parameters **number** – How many fingers to return.

Returns The *number* of instances of `Finger`.

import_nodes (*nodes_list*)
Import a list of nodes.

Receives a list of tuples, typically from a foreign node exporting their list, and adds those nodes to the FingerSpace.

Data is expected to be (ip address, port, public key[, ident]) The ident is optional.

Parameters **nodes** – List of nodes to import.

put (*ip_address*, *listening_port*, *public_key*, *existing_ident=''*)
Place a new node into the Finger Space.

Expect a `FingerError` exception be raised if the data passed in is not valid. Also expect a `HashMissmatchError` exception if the generated ident does not match one passed in, do try to pass this data in to maintain integrity.

Parameters

- **ip_address** – IP address of the node.
- **listening_port** – Listening port of the node.
- **public_key** – Public Key of the node in binary format.

remove (*ident*)
Delete a Node Finger from the FingerSpace

Parameters **ident** – ident of the Finger to remove.

Returns True if succesfully removed, false if otherwise.

`distrim.fingerspace.`**finger_type_test** (*ip_address*, *listening_port*, *public_key*)
Tests three values for correct type and format.

Parameters

- **ip_address** – IP address of the node.
- **listening_port** – Listening port of the node.
- **public_key** – Public Key of the node.

Returns True if parameters are valid, else raises a *FingerError* exception.

`distrim.fingerspace.`**generate_hash** (*ip_address*, *listening_port*, *public_key*)
Creates the identifying hash.

The hash is generated using the sha256 function. The IP address, listening port, and the public key are concatenated together into a single string. The string is hashed using the sha256 function.

For demonstration purposes the length of the hash is reduced to 2 bytes.

Parameters

- **ip_address** – IP address of the node.

- **listening_port** – Listening port of the node.
- **public_key** – Public Key of the node in binary format.

Returns String representation of an MD5 hex hash.

`distrim.fingerspace.`**`h2i`**`(`*hex_string*`)`

Converts a hexadecimal string into an integer.

For example: '2e' -> 46

This is used since the key for entries in the fingerspace is the Finger ident represented as a number.

Parameters **hex_string** – The string to convert.

Returns Integer representation of the hex string.

E.3.3 Node

class `distrim.node.`**`Node`**`(`*local_ip*, *local_port=2000*, *log_ip=''*, *log_port=1999*`)`

Representation of a single Node in the DistrIM network.

`__init__``(`*local_ip*, *local_port=2000*, *log_ip=''*, *log_port=1999*`)`

A node within the peer-to-peer network.

Parameters

- **local_ip** – IP address of this node.
- **local_port** – Listening port of this node.
- **log_ip** – IP address of a remote logger.
- **log_port** – Port of the remote logger.

`send_message``(`*recipient*, *message*`)`

Send a message.

Parameters

- **recipient** – Ident of the recipient.
- **message** – The message to send.

`start``(`*remote_ip=''*, *remote_port=2000*`)`

Parameters

- **remote_ip** – IP address of a remote note to bootstrap against.
- **remote_port** – Listening port of the remote node.

`stop``()`

Stop the node and exit from the network.

E.3.4 Protocol

DistrIM has a clearly defined protocol which nodes must adhere to. Nodes following the protocol should find their messages remain secure.

class `distrim.protocol.`**`Announcer`**(*log*, *local_finger*, *local_keys*, *foreign_finger*)
Handler for announcing ourselves to foreign nodes.

`__init__`(*log*, *local_finger*, *local_keys*, *foreign_finger*)

Parameters

- **log** – Logger instance to output to.
- **fingerspace** – The FingerSpace instance of this node.
- **local_finger** – The Finger of this node.
- **local_keys** – The CipherWrapper of this node.
- **foreign_finger** – The Finger of the foreign node.

`announce`()
Send local finger information to a remote node.

class `distrim.protocol.`**`Boostrapper`**(*log*, *fingerspace*, *local_finger*, *local_keys*)
Handle bootstrap and rendezvous.

Protocol Handler specialised for bootstrapping and rendezvousing with other nodes in the network.

`__init__`(*log*, *fingerspace*, *local_finger*, *local_keys*)

Parameters

- **log** – Logger instance to output to.
- **fingerspace** – The FingerSpace instance of this node.
- **local_finger** – The Finger of this node.
- **local_keys** – The CipherWrapper of this node.

`_init_connection`(*remote_address*)
Establish connection and setup this object.

`_setup`(*foreign_info*)
Set necessary local variables after initial connection.

Since little is known about the foreign node prior to the creation of this object, it's necessary to fill in information about the foreign node before two-way message passing can happen.

`announce`()
Make presence of this node known to others.

bootstrap(*remote_address*)
Perform bootstrap procedure.

The very first action a node will perform is its bootstrap procedure, during which the node will rendezvous with a bootstrap node, an existing node in the network, and attain a list of nodes.

Parameters **remote_address** – IP and Port tuple of bootstrap node.

class distrim.protocol.**ConnectionHandler**
An abstract class with common connection functionality.

ConnectionHandler holds some common functionality used for the connection of nodes to one another.

Messages can be sent between the local node and the foreign node through an instance of this class. The instance will take care of pickling and encrypting the messages. Transmission is achieved by a SocketWrapper.

Pickled messages are created with the *cPickle* module.

When the message is unpickled, a tuple of length 4 will be attained with the following attributes:

- The sender's information.
- The message type.
- The message parameters.
- The padding.

The sender's information will contain the sender's finger and any nonces used for the transaction which are checked for consistency. The padding is used for cryptographic scrambling and is discarded.

_verify_foreign(*sender_info*)
Verify the foreign node

_verify_message(*msg_type*, *parameters*, *expected=None*)
Check message for consistency

close()
Terminate the connection

connect(*remote_address=None*)
Establish connection with foreign node

package(*message_type*, *parameters*)
Construct a message for sending to a foreign node.

Parameters

- **message_type** – Type of message from the Protocol class.
- **parameters** – Parameters of the message, as a dict.

receive (*expected=None*)
Receive a message from the foreign node.

Returns A message type, and its parameters

send (*message_type*, *parameters*)
Construct a message and send it.

Parameters

- **message_type** – The type of message defined in `Protocol`
- **parameters** – Parameters of the message.

unpack (*cryptic_data*)
Unpack data sent to this node by a foreign node.

class `distrim.protocol.`**IncomingConnection** (*log*, *sock*, *addr*, *fingerspace*, *local_finger*, *local_keys*)
Protocol Handler for communication with foreign nodes.

The methods of this class define procedures for dealing with connections from foreign nodes.

__init__ (*log*, *sock*, *addr*, *fingerspace*, *local_finger*, *local_keys*)

Parameters

- **log** – Logger instance to output to.
- **sock** – socket object of the incoming connection.
- **addr** – address of the connecting node.
- **fingerspace** – The FingerSpace instance of this node.
- **local_finger** – The Finger of this node.
- **local_keys** – The CipherWrapper of this node.

_is_bootstrap_request (*data*)
Determine if this node is trying to rendezvous.

Nodes that have not joined the network know of no other node or their public key, so they will send their finger information unencrypted to a bootstrap node. This attempts to load that data, if it fails then it is assumed that it is an encrypted message from an existing node and will be handled appropriately.

Parameters **data** – Raw data string received from the foreign node.

Returns True if this is a bootstrap request, else False.

_peel_onion_layer (*package*)
Strips a layer from a message package

_rendezvous ()
Accept a new node into the network by sharing our finger table.

`handle`()
Perform handling of the incoming connection.

Receives data from the foreign node and deciphers it, this will call one of the relevant handlers to deal with the connection based on what the message type is.

`handle_announcement`(*params*)
Put node information in the FingerSpace

`handle_leaver`(*params*)
Remove a foreign node from network

`handle_relay`(*params*)
Relay package from one node to another

class `distrim.protocol.`**`Leaver`**(*log*, *local_finger*, *local_keys*, *foreign_finger*)
Handler for announcing departure to foreign nodes.

`__init__`(*log*, *local_finger*, *local_keys*, *foreign_finger*)

Parameters

- **log** – Logger instance to output to.
- **fingerspace** – The FingerSpace instance of this node.
- **local_finger** – The Finger of this node.
- **local_keys** – The CipherWrapper of this node.
- **foreign_finger** – The Finger of the foreign node.

`leave`()
Send local finger information to a remote node.

class `distrim.protocol.`**`MessageHandler`**(*log*, *fingerspace*, *local_finger*, *local_keys*, *foreign_finger=None*)
Protocol Handler for outgoing communication with foreign nodes.

The methods of this class define procedures for dealing with connections established locally to transmit to foreign nodes.

`__init__`(*log*, *fingerspace*, *local_finger*, *local_keys*, *foreign_finger=None*)

Parameters

- **log** – Logger instance to output to.
- **fingerspace** – The FingerSpace instance of this node.
- **local_finger** – The Finger of this node.
- **local_keys** – The CipherWrapper of this node.
- **foreign_finger** – The Finger of the foreign node.

`_build_message`(*recipient*, *message*)
Construct the final message package received by the recipient.

Parameters

- **recipient** – Finger of the recipient.
- **message** – Textual message for the recipient to receive.

`_build_onion`(*recipient*, *package*)
Construct the onion package

`send_message`(*recipient*, *message*)
Send message.

class `distrim.protocol.`**`Protocol`**
Protocol Message Definitions.

Protocol messages should be in alphabetic order and the values should be 4 characters in length.

E.3.5 Command Line User Interface

CLI Documentation

class `distrim.ui_cl.`**`CommandLine`**(*node_params*)
A Command Line Interface (CLI) for controlling an instance of a `Node`, including viewing node information and using the network to send messages.

`__init__`(*node_params*)

Parameters **node_params** – Dictionary of parameters for `Node`

`accept_commands`()
Prompt for command input and execute a command.

`cmd_help`(*params*)
Input Command: Display Help

`cmd_print`(*params*)
Input Command: Print Data to terminal.

`cmd_quit`(*params*)
Input Command: Terminate the node and exit.

`cmd_send`(*params*)
Input Command: Send a message

`enter`(*boot_params*)
Start node and accept commands in a loop.

Parameters **boot_params** – Settings required for starting the node.

`exec_command`(*command*)
Parse command and attempt to execute it.

`get_command`()
Receive input from the command prompt.

parse_command(*command*)
Parse an input command and return a handler.

distrim.ui_cl.**run_application**(*args*)
Entry point for the program.

Initiates the application and fetches any configuration from the program arguments.

Parameters **args** – Arguments collected by **:module:‘argparse‘**.

E.4 Assets and Utilities

E.4.1 Errors

Collection of custom Exception types that are defined for use in DistrIM.

exception distrim.assets.errors.**AuthError**
Raised if authentication with a foreign node fails.

exception distrim.assets.errors.**CipherError**
Raised by improper use of the CipherWrap class.

exception distrim.assets.errors.**FingerError**
Raised by creating a finger with invalid data

exception distrim.assets.errors.**FingerSpaceError**
Raised if an error occurs in the FingerSpace

exception distrim.assets.errors.**HashMissmatchError**(*addr*, *port*, *hash_gen*, *hash_bad*)
Raised if two idents, which should match, do not.

__init__(*addr*, *port*, *hash_gen*, *hash_bad*)

Parameters

- **addr** – Node address.
- **port** – Node Port.
- **hash_gen** – Generated ident.
- **hash_bad** – Given ident.

exception distrim.assets.errors.**NetInterfaceError**
Raised if failure getting local IP address.

exception distrim.assets.errors.**ProcedureError**
Raised during communications if data sent at incorrect time.

exception distrim.assets.errors.**ProtocolError**
Raised during communications if invalid data is sent or received.

exception `distrim.assets.errors.`**`SockWrapError`**

Raised by improper use of the SocketWrapper class or to wrap the rather ghastly *socket.error* exception.

E.4.2 Utilities

Miscelanious utility functions and classes used in DistrIM.

class `distrim.utils.utilities.`**`CipherWrap`**(*cipher*)

Wrap an RSA Cipher Instance

`__init__`(*cipher*)

Parameters **cipher** – An RSA key, or an RSA instance.

`decrypt`(*cryptic_data*)

Decrypt a packet of data.

Parameters **cryptic_data** – The encrypted data to decrypt.

Returns The decrypted data.

`encrypt`(*data*, *split_size=128*)

Encrypt a packet of data.

Note that this data must be a string no longer than 128 bytes.

Parameters **data** – The data to encrypt.

Returns The encrypted data.

`export`(*text=False*, *key_type=0*)

Export the RSA key as a string.

By default, this just exports a public key in DER format for use in fingers.

If the private key is requested when this instance only has a public key, then a `CipherError` is thrown.

Parameters

- **text** – If True, format the string for humans.
- **key_type** – Key type to export. If 0, public; if 1, private; if 2, export public and private key.

Returns The exported key.

class `distrim.utils.utilities.`**`SocketWrapper`**(*sock=None*, *remote_address=None*, *timeout=15*)

Socket interface for communication with foreign nodes.

This class wraps around a `socket.socket` object. It provides the ability to send and receive packed data, packing it with the length to ensure all data is received.

If the socket is not connected, use the `connect()` method to establish the connection.

`__init__`(*sock=None*, *remote_address=None*, *timeout=15*)
Create the wrapper for the sockets.

Note: You can pass in a socket or an address, if you pass in a connected socket, the remote address will be ignored.

Parameters

- **sock** – the *socket* object. If None, a socket is created using the default values.
- **remote_address** – IP and Port of the remote host.
- **timeout** – the timeout value of the socket, how long it will pend waiting for a remote response.

`_test_connection`()
Test if connected, raise exception if not.

`close`()
Close connection with the foreign node.

`connect`(*remote_address=None*)
Connect the socket to the remote address.

Parameters **remote_address** – IP and Port of the remote host.

`is_connected`()
Determines if the socket is connected or not. :return: True if it is, False if it isn't.

`receive`(*read_length=1024*)
Receive data from a foreign node via its socket.

Parameters **read_length** – How many bytes to read at a time.

`send`(*data*)
Send data to the foreign node via its socket.

Parameters **data** – The data packet to send.

`distrim.utils.utilities.`**`format_elapsed`**(*delta*)
Format a `datetime.timedelta` object into a string.

`distrim.utils.utilities.`**`generate_padding`**(*min_length=64*, *max_length=512*)
Create a padding string for use in a cryptographic message

Generate a random string, of random characters, of a random length for padding secure messages.

Parameters

- **min_length** – Minimum length of the padding.
- **max_length** – Maximum length of the padding.

Returns The padding.

`distrim.utils.utilities.`**`get_local_ip`**(*address_type=2*)

Determine local IP address of node from its interface IP.

Parameters **address_type** – Any address type from the *AF_** values in the *netifaces* module. Default *AF_INET* for IPv4 addresses.

`distrim.utils.utilities.`**`split_address`**(*address*)

Transform IPv4 address and port into a *string* and *int* tuple.

Parameters **address** – The string format of the input address.

Returns tuple of string of the IP or hostname, and port as an int.

`distrim.utils.utilities.`**`split_chunks`**(*seq*, *part_size=128*)

Split a sequence into parts.

Parameters

- **seq** – The sequence to split.
- **part_size** – Size of the parts.

Returns Generator function that yields chunks.

APPENDIX F

CODE

This is the code listing for DistrIM.

The directory structure shows the structure of all code files within the product. Listings have only been provided for the classes considered significant.

Note that all code associated with the product is available on the product CD.

F.1 Directory Structure

```
distrim/assets/errors.py
distrim/assets/text.py
distrim/assets/__init__.py
distrim/connections.py
distrim/fingerspace.py
distrim/node.py
distrim/protocol.py
distrim/ui_cl.py
distrim/unit_tests/test_fingerspace.py
distrim/unit_tests/test_protocol.py
distrim/unit_tests/__init__.py
distrim/utils/config.py
distrim/utils/logger.py
distrim/utils/unit_tests/test_utilities.py
distrim/utils/unit_tests/__init__.py
distrim/utils/utilities.py
distrim/utils/__init__.py
distrim/__init__.py
main_distrim.py
```

F.2 Code Listing

F.2.1 main_distrim.py

```python

"""
    Entry point to DistrIM, this should be executed on the command line within
    the Python environment.
"""

import argparse

from distrim.ui_cl import run_application
from distrim.assets.text import CMD_DESCRIPTION, CMD_EPILOG
from distrim.utils.utilities import split_address


def init():
    """
    Entry point for the DistrIM application.

    Takes arguments from the command line and creates a Command Line Interface
    with a DistrIM node.
    """
    # ArgParse docs: https://docs.python.org/dev/library/argparse.html
    parser = argparse.ArgumentParser(
        description=CMD_DESCRIPTION, epilog=CMD_EPILOG,
        formatter_class=argparse.RawTextHelpFormatter)
    parser.add_argument('-p', '--listen-on', type=int,
                        help='listening port for this node')
    parser.add_argument('-b', '--bootstrap', type=split_address,
                        help='Address for a bootstrap node in form IP:Port.')
    parser.add_argument('-l', '--logger', type=split_address,
                        help='Address for a remote logger in form IP:Port.')

    args = parser.parse_args()
    run_application(args.__dict__)


# Program entry point
if __name__ == '__main__':
    init()
```

F.2.2 distrim/connections.py

```python

"""
    Connections Manager
"""


import socket
import traceback

from time import sleep
from threading import Thread
from thread_pool import ThreadPool

from .protocol import IncomingConnection, MessageHandler, Boostrapper, Leaver
from .utils.config import CFG_THREAD_POOL_LENGTH, CFG_LISTENING_QUEUE


class ConnectionsManager(object):
    """
    The ConnectionsManager class is responsible for all incoming connections
```

```
        from other nodes.
        """
    def __init__(self, parent_log, local_ip, local_port,
                 fingerspace, finger, keys):
        """
        :param parent_log:
        """
        self.log = parent_log.getChild(__name__.rpartition('.')[2])
        self.local_ip = local_ip
        self.local_port = local_port
        self.fingerspace = fingerspace
        self.local_finger = finger
        self.local_keys = keys
        self._running = False

        # Listener
        self._pool = ThreadPool(CFG_THREAD_POOL_LENGTH)
        self._thread = Thread(target=self._listen, name='Thread-Listener')
        self._thread.daemon = True
        self._sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

        # Cleaner
        self._cleaner = Thread(target=self._cleaning, name='Thread-Cleaner')
        self._cleaner.daemon = True

        # Some nice stats, because why not
        self.count_conn_success = 0
        self.count_conn_failure = 0

    def start(self):
        """Begin the listening and cleaning threads of this node."""
        self._running = True
        self._sock.bind((self.local_ip, self.local_port))
        self._sock.listen(CFG_LISTENING_QUEUE)
        self._thread.start()
        self._cleaner.start()
        self.log.info("Listening for connections on %s:%d", self.local_ip,
                      self.local_port)

    def stop(self):
        """Stop listening for connections and end the listening thread."""
        self._running = False
        try:
            self._sock.shutdown(socket.SHUT_RD)
            self._sock.close()
        except socket.error:
            pass

        # Announce leaving to everyone.
        for finger in self.fingerspace.get_all():
            leaver = Leaver(self.log, self.local_finger, self.local_keys,
                            finger)
            leaver.leave()

        while not self._pool.out_queue.empty():
            sleep(0.2)

    def bootstrap(self, remote_ip, remote_port):
        """
        Establish the first connection in the network.
        """
        connection = Boostrapper(self.log, self.fingerspace, self.local_finger,
                                 self.local_keys)
        connection.bootstrap((remote_ip, remote_port))

    def send_message(self, recipient, message):
        """
```

```python
        Send a message via relays.

        :param recipient: Finger of the node to send the message to.
        :param message: Plaintext message to send.
        """
        postman = MessageHandler(
            self.log, self.fingerspace, self.local_finger, self.local_keys)
        postman.send_message(recipient, message)

    def pool_new_connection(self, sock, address):
        """
        Handle incoming connection, puts socket into seperate thread.
        """
        self._pool.add_task(self.accept_new_connetion, sock, address)

    def accept_new_connetion(self, sock, address):
        """
        Handle incoming connections

        :param sock: The socket of the incoming connection.
        :param address: Address of the connecting node.
        """
        try:
            self.log.info('New Connection from: %s', address)
            connection = IncomingConnection(
                self.log, sock, address, self.fingerspace, self.local_finger,
                self.local_keys)
            connection.handle()
            connection.close()
        except Exception as exc:  # pylint: disable=broad-except
            traceback.print_exc()
            self.log.error("Exception occured during connection with %s:\n%s",
                           address, exc.message)
            return False
        return True

    def _listen(self):
        """
        Listen for incoming connections.

        This method is the target of `self._thread`
        """
        while self._running:
            try:
                sock, addr = self._sock.accept()
                self.pool_new_connection(sock, addr)
            except socket.error as exc:
                if exc.errno != 22:
                    self.log.error("Socket error: %s", exc.strerror)
        self.log.debug("Listening thread stopped.")

    def _cleaning(self):
        """
        Removes completed connection results from the thread pool.

        This method is the target of `self._cleaner`
        """
        while self._running:
            result = self._pool.get_task()
            try:
                self._pool.out_queue.task_done()
            except ValueError:
                self.log.error("_pool.out_queue error")
            if result:
                self.count_conn_success += 1
            else:
                self.count_conn_failure += 1
```

```
            # It'll be satisfactory for the time being to pend on get_task
            # Hopefully errors will be rare
            try:
                while not self._pool.err_queue.empty():
                    self._pool.err_queue.get_nowait()
            except Exception as exc:  # pylint: disable=broad-except
                self.log.error("Cleaning errors: %s", exc.message)
        self.log.debug("Cleaning thread stopped.")
```

F.2.3 distrim/fingerspace.py

```

"""
    Finger Space, stores information about other nodes
"""

import random

from hashlib import sha256
from threading import Semaphore
from Crypto.PublicKey import RSA

from .assets.errors import (HashMissmatchError, FingerSpaceError,
                            FingerError)
from .utils.utilities import SocketWrapper, CipherWrap


class Finger(object):
    """
    Contains identifying information unique to a single node.

    This class represents the information identifying a particular Node in the
    network. Provides some functionalty for connecting and communicating with
    the node.

    Four attributes are stored:
     - The ident of the node.
     - The IP address of the node.
     - The listening port of node.
     - The public key of the node.

    These are the attributes needed for communication between nodes.

    On instantiation, an optional identifier value can also be passed in; the
    ident is calculated anyway but if given then it can be validated for
    authenticity.
    """
    def __init__(self, ip_address, listening_port, public_key, ident=''):
        """
        :param ip_address: IP address of the node.
        :param listening_port: Listening port of the node.
        :param public_key: Public key of the node.
        :param ident: Hash value representing the identity of the node.
        """
        new_ident = generate_hash(ip_address, listening_port, public_key)
        if ident and (ident != new_ident):
            raise HashMissmatchError(ip_address, listening_port,
                                     new_ident, ident)
        self.addr = ip_address
        self.port = listening_port
        self.key = public_key
        self.ident = new_ident
        # Combined values
        self.address = (self.addr, self.port)
        self.values = (self.addr, self.port, self.key)
        self.all = (self.addr, self.port, self.key, self.ident)
```

```
    def __eq__(self, other):
        """
        Check if this finger is equal to another.

        :param other: The other finger.
        """
        # Type check, attribute check
        if isinstance(other, Finger) and self.__dict__ == other.__dict__:
            return True
        return False

    def __repr__(self):
        """
        Representation of this object by text.
        """
        return "<Fingerspace.Finger %s @ %s:%d>" % (self.ident, self.addr,
                                                    self.port)

    def get_socket(self):
        """
        Get a socket object connected to this node.

        :return: A :class:`SocketWrapper` instance with internal address
            defined, but not connected.
        """
        return SocketWrapper(remote_address=self.address)

    def get_cipher(self):
        """
        Get an RSA cipher for message encryption.

        Returns an instance of an RSA cipher of the Public Key of this Node.

        :return: RSA Public Key instance of type :class:`CipherWrap`.
        """
        return CipherWrap(self.key)


class FingerSpace(object):
    """
    The FingerSpace class is responsible for storing information about nodes.
    Access to the Key Space is managed through this class.
    """
    def __init__(self, parent_log, local_finger):
        """
        :param parent_log: logger object from Node instance.
        :param local_finger: The finger for this node.
        """
        self.log = parent_log.getChild(__name__.rpartition('.')[2])
        self.local_finger = local_finger
        self.access = Semaphore()
        self._keyspace = {}
        random.seed()

        # Some nice stats
        self.count_added = 0
        self.count_removed = 0

    def __len__(self):
        """FingerSpace length, the number of keys stored"""
        with self.access:
            return len(self._keyspace)

    def import_nodes(self, nodes_list):
        """
        Import a list of nodes.
```

```
        Receives a list of tuples, typically from a foreign node exporting
        their list, and adds those nodes to the FingerSpace.

        Data is expected to be (ip address, port, public key[, ident])
        The ident is optional.

        :param nodes: List of nodes to import.
        """
        for values in nodes_list:
            try:
                self.put(*values)  # Star-input allows us to use 3 or 4 args
            except FingerError as exc:
                self.log.error("Error importing finger: %s", exc.message)

    def export_nodes(self):
        """
        Export a list of all nodes.

        Exports a list of all nodes in tuple format for serialising and sending
        to foreign nodes.

        Data is exported as (ip address, port, public key, ident)

        :return: A list of tuples with the data from the 'all' attribute.
        """
        with self.access:
            return [finger.all for finger in self._keyspace.itervalues()]

    def get_all(self):
        """
        Gets a list of all fingers.
        """
        with self.access:
            return self._keyspace.values()

    def get(self, ident):
        """
        Retrieve a Node Finger with the ident.

        :param ident: ident of the Finger to fetch.
        :return: Finger of the node, or `None` if node not found.
        """
        with self.access:
            return self._keyspace.get(h2i(ident), None)

    def put(self, ip_address, listening_port, public_key, existing_ident=''):
        """
        Place a new node into the Finger Space.

        Expect a :class:`FingerError` exception be raised if the data passed
        in is not valid. Also expect a :class:`HashMissmatchError` exception
        if the generated ident does not match one passed in, do try to pass
        this data in to maintain integrity.

        :param ip_address: IP address of the node.
        :param listening_port: Listening port of the node.
        :param public_key: Public Key of the node in binary format.
        """
        finger = Finger(ip_address, listening_port,
                        public_key, existing_ident)

        if self.local_finger == finger:
            self.log.warning("Can't place local finger in FingerSpace")
            return

        ident = h2i(finger.ident)
```

```python
        with self.access:
            if ident not in self._keyspace:
                self._keyspace[ident] = finger
                self.count_added += 1
            else:
                if not self._keyspace[ident] == finger:
                    self.log.warning(
                        "Attempted adding non-matching finger with matching "
                        + "ident %s.", finger.ident)

    def remove(self, ident):
        """
        Delete a Node Finger from the FingerSpace

        :param ident: ident of the Finger to remove.
        :return: True if succesfully removed, false if otherwise.
        """
        try:
            with self.access:
                self._keyspace.pop(h2i(ident))
            self.count_removed += 1
            return True
        except KeyError:
            return False

    def get_random_fingers(self, number):
        """
        Get random fingers.

        :param number: How many fingers to return.
        :return: The *number* of instances of :class:`Finger`.
        """
        with self.access:
            idents = self._keyspace.keys()
            if not self._keyspace:
                raise FingerSpaceError("DHT is empty.")

        if number < 1:
            raise ValueError("Number of keys must be positive")
        if len(idents) < number:
            number = len(idents)

        route = []

        with self.access:
            for _ in xrange(number):
                key = random.choice(idents)
                route.append(self._keyspace[key])
                idents.remove(key)
        return route


def generate_hash(ip_address, listening_port, public_key):
    """
    Creates the identifying hash.

    The hash is generated using the sha256 function. The IP address, listening
    port, and the public key are concatenated together into a single string.
    The string is hashed using the sha256 function.

    For demonstration purposes the length of the hash is reduced to 2 bytes.

    :param ip_address: IP address of the node.
    :param listening_port: Listening port of the node.
    :param public_key: Public Key of the node in binary format.

    :return: String representation of an MD5 hex hash.
```

```
    """
    finger_type_test(ip_address, listening_port, public_key)
    concated = "%s%d%s" % (ip_address, listening_port, public_key)
    ash = sha256(concated)
    return ash.hexdigest()[:4]


def finger_type_test(ip_address, listening_port, public_key):
    """
    Tests three values for correct type and format.

    :param ip_address: IP address of the node.
    :param listening_port: Listening port of the node.
    :param public_key: Public Key of the node.

    :return: True if parameters are valid, else raises a `FingerError`
        exception.
    """
    # Test Types
    if not isinstance(ip_address, str):
        raise FingerError("ip_address must be a string")
    if not isinstance(listening_port, int):
        raise FingerError("listening_port must be an int")
    if not isinstance(public_key, str):
        raise FingerError("public_key must be a string")

    # Test values
    if len(ip_address.split('.')) != 4:
        raise FingerError("Invalid IPv4 address: '%s'" % (ip_address,))
    if listening_port > 65535 or listening_port < 1:
        raise FingerError("invalid port number '%d'. " % (listening_port,)
                          + "Must be between 1 and 65535")

    if public_key.startswith('-----BEGIN'):
        raise FingerError("public_key must be in binary format")

    try:
        key = RSA.importKey(public_key)
        if key.has_private():
            raise FingerError("!!!This is a private key, not public!!!")
    except (ValueError, IndexError) as exc:
        raise FingerError("public_key is not valid:\n%s" % exc.message)

    return True


def h2i(hex_string):
    """
    Converts a hexadecimal string into an integer.

    For example: '2e' -> 46

    This is used since the key for entries in the fingerspace is the Finger
    ident represented as a number.

    :param hex_string: The string to convert.
    :return: Integer representation of the hex string.
    """
    return int(hex_string, 16)
```

F.2.4 distrim/node.py

```

"""
    A DistrIM Node.
"""

```

```python

from Crypto.PublicKey import RSA
from datetime import datetime as dto

from .connections import ConnectionsManager
from .fingerspace import Finger, FingerSpace

from .utils.config import CFG_LISTENING_PORT, CFG_LOGGER_PORT, CFG_KEY_LENGTH
from .utils.logger import create_logger
from .utils.utilities import CipherWrap


class Node(object):
    """
    Representation of a single Node in the DistrIM network.
    """
    def __init__(self, local_ip, local_port=CFG_LISTENING_PORT, log_ip='',
                 log_port=CFG_LOGGER_PORT):
        """
        A node within the peer-to-peer network.

        :param local_ip: IP address of this node.
        :param local_port: Listening port of this node.
        :param log_ip: IP address of a remote logger.
        :param log_port: Port of the remote logger.
        """
        self.local_ip = local_ip
        self.local_port = local_port
        # Cryptographic Settings
        # https://pythonhosted.org/pycrypto/
        crypto_key = RSA.generate(CFG_KEY_LENGTH)
        self.keys = CipherWrap(crypto_key)

        # Identity
        self.finger = Finger(local_ip, local_port,
                             self.keys.export(text=False, key_type=0))

        # Get Logging!
        # __name__ is distrim.node
        self.log = create_logger(__name__, log_ip, log_port,
                                 ident=self.finger.ident)

        self.fingerspace = FingerSpace(self.log, self.finger)
        self.conn_manager = ConnectionsManager(self.log, local_ip, local_port,
                                               self.fingerspace, self.finger,
                                               self.keys)

    def start(self, remote_ip='', remote_port=CFG_LISTENING_PORT):
        """
        :param remote_ip: IP address of a remote note to bootstrap against.
        :param remote_port: Listening port of the remote node.
        """
        self.log.info("Node started %s @ %s:%d", self.finger.ident,
                      self.local_ip, self.local_port)
        self.start_time = dto.now()
        self.conn_manager.start()
        if remote_ip:
            self.log.info("Boostrapping to %s:%d", remote_ip, remote_port)
            self.conn_manager.bootstrap(remote_ip, remote_port)

    def stop(self):
        """
        Stop the node and exit from the network.
        """
        self.log.info("Node Stopping...")
        self.conn_manager.stop()

```

```python
    def send_message(self, recipient, message):
        """
        Send a message.

        :param recipient: Ident of the recipient.
        :param message: The message to send.
        """
        rec = self.fingerspace.get(recipient)
        if not rec:
            self.log.error("No such node: %s", recipient)
            return
        return self.conn_manager.send_message(rec, message)
```

F.2.5 distrim/protocol.py

```python

"""
    Protocol, handlers for connections with other nodes
"""

from hashlib import md5

import pickle
from pickle import UnpicklingError

from .fingerspace import Finger
from .assets.errors import (ProtocolError, ProcedureError, AuthError,
                            SockWrapError)
from .utils.config import CFG_PICKLE_PROTOCOL, CFG_PATH_LENGTH
from .utils.utilities import SocketWrapper, generate_padding


class Protocol(object):
    """
    Protocol Message Definitions.

    Protocol messages should be in alphabetic order and the values should be
    4 characters in length.
    """
    Announce = "ANNO"
    Message = "MESG"
    Ping = "PING"
    Pong = "PONG"
    Quit = "QUIT"
    Relay = "RELY"
    Welcome = "WELC"
    ALL = [Announce, Message, Ping, Pong, Quit, Relay, Welcome]


class ConnectionHandler(object):
    """
    An abstract class with common connection functionality.

    ConnectionHandler holds some common functionality used for the connection
    of nodes to one another.

    Messages can be sent between the local node and the foreign node through an
    instance of this class. The instance will take care of pickling and
    encrypting the messages. Transmission is achieved by a SocketWrapper.

    Pickled messages are created with the *cPickle* module.

    When the message is unpickled, a tuple of length 4 will be attained with
    the following attributes:

     - The sender's information.
     - The message type.
```

```
     - The message parameters.
     - The padding.

    The sender's information will contain the sender's finger and any nonces
    used for the transaction which are checked for consistency. The padding
    is used for cryptographic scrambling and is discarded.
    """
    def __init__(self):
        raise NotImplementedError("ConnectionHandler is abstract!")
        # pylint: disable=no-member

    def send(self, message_type, parameters):
        """
        Construct a message and send it.

        :param message_type: The type of message defined in :class:`Protocol`
        :param parameters: Parameters of the message.
        """
        self._verify_message(message_type, parameters)
        cryptic_data = self.package(message_type, parameters)
        self.conn.send(cryptic_data)

    def receive(self, expected=None):
        """
        Receive a message from the foreign node.

        :return: A message type, and its parameters
        """
        cryptic_data = self.conn.receive()  # Receive foreign data

        try:
            foreign, message_type, parameters = self.unpack(cryptic_data)
        except ValueError:
            raise ProtocolError("Error unpacking received data.")
        self._verify_foreign(foreign)
        self._verify_message(message_type, parameters, expected)
        return message_type, parameters

    def package(self, message_type, parameters):
        """
        Construct a message for sending to a foreign node.

        :param message_type: Type of message from the Protocol class.
        :param parameters: Parameters of the message, as a dict.
        """
        if message_type not in Protocol.ALL:
            raise ProtocolError("Invalid protocol message type '%s'"
                                % (message_type,))
        if type(parameters) is not dict:
            raise ProtocolError("Message parameters must be in a dictionary.")

        msg = (self.local_finger.all, message_type, parameters)

        data = pickle.dumps(msg, protocol=CFG_PICKLE_PROTOCOL)
        data_pack = data + generate_padding()

        cryptic_data = self.foreign_key.encrypt(data_pack)
        return cryptic_data

    def unpack(self, cryptic_data):
        """
        Unpack data sent to this node by a foreign node.
        """
        data = self.local_keys.decrypt(cryptic_data)

        try:
            foreign, msg_type, params = pickle.loads(data)
```

```
        except UnpicklingError as exc:
            self.log.error("Unpickling error, %s", exc.message)
            self.log.error("Decrypted hash: %s", md5(data).hexdigest())
            raise ProtocolError("Couldn't de-serialise: %s" % (exc.message,))

        return foreign, msg_type, params

    def _verify_foreign(self, sender_info):
        """Verify the foreign node"""
        sender_finger = Finger(*sender_info)
        try:
            if self.foreign_finger != sender_finger:
                self.log.warning("Authentication error with %s",
                                 self.foreign_finger.ident)
                raise AuthError("Info of foreign not match of locally stored")
        except AttributeError:
            self.log.debug("Authenticating new connection...")
            self.foreign_finger = sender_finger
            self.foreign_key = sender_finger.get_cipher()
            self.fingerspace.put(*sender_info)

    def _verify_message(self, msg_type, parameters, expected=None):
        """Check message for consistency"""
        if msg_type not in Protocol.ALL:
            raise ProtocolError("Received message not valid protocol")

        if expected and expected != msg_type:
            raise ProcedureError("Expected message type '%s' but got '%s'" %
                                 (expected, msg_type))

        for key in parameters.keys():
            if key.upper() != key:
                raise ProtocolError("Invalid key in parameters '%s'." % (key,))

    def connect(self, remote_address=None):
        """Establish connection with foreign node"""
        if remote_address:
            self.conn.connect(remote_address)
        elif self.foreign_finger:
            self.conn.connect(self.foreign_finger.address)
        else:
            raise ProtocolError("No address to connect to.")

    def close(self):
        """Terminate the connection"""
        try:
            self.conn.close()
        except SockWrapError as exc:
            self.log.error("Error closing socket: %s", exc.message)
        # pylint: enable=no-member


class Boostrapper(ConnectionHandler):
    """
    Handle bootstrap and rendezvous.

    Protocol Handler specialised for bootstrapping and rendezvousing with
    other nodes in the network.
    """
    def __init__(self, log, fingerspace, local_finger, local_keys):
        """
        :param log: Logger instance to output to.
        :param fingerspace: The FingerSpace instance of this node.
        :param local_finger: The Finger of this node.
        :param local_keys: The CipherWrapper of this node.
        """
        self.log = log.getChild('bootstrapper')
```

```python
        self.conn = SocketWrapper()
        self.fingerspace = fingerspace
        self.local_finger = local_finger
        self.local_keys = local_keys

    def _setup(self, foreign_info):
        """
        Set necessary local variables after initial connection.

        Since little is known about the foreign node prior to the creation of
        this object, it's necessary to fill in information about the foreign
        node before two-way message passing can happen.
        """
        self.foreign_finger = Finger(*foreign_info)
        self.foreign_key = self.foreign_finger.get_cipher()
        self.fingerspace.put(*self.foreign_finger.all)

    def _init_connection(self, remote_address):
        """
        Establish connection and setup this object.
        """
        self.conn.connect(remote_address)
        self.log.debug("Bootstrap connection established.")
        boot_package = pickle.dumps(self.local_finger.all,
                                    protocol=CFG_PICKLE_PROTOCOL)
        self.conn.send(boot_package)
        self.log.debug("Bootstrap package sent.")

        # Expect back a welcome message.
        cryptic_data = self.conn.receive()  # Receive foreign data
        foreign, message_type, parameters = self.unpack(cryptic_data)
        if message_type != Protocol.Welcome:
            raise ProcedureError("Expected welcome from bootstrap node.")
        self._setup(foreign)
        return parameters

    def bootstrap(self, remote_address):
        """
        Perform bootstrap procedure.

        The very first action a node will perform is its bootstrap procedure,
        during which the node will rendezvous with a bootstrap node, an
        existing node in the network, and attain a list of nodes.

        :param remote_address: IP and Port tuple of bootstrap node.
        """
        welcome_params = self._init_connection(remote_address)
        # We will add your technological distinctiveness to our own.
        nodes_list = welcome_params.get('NODES')
        if nodes_list:
            self.fingerspace.import_nodes(nodes_list)
            self.announce()
        self.log.info("SUCCESS! Rendezvous occured.")

    def announce(self):
        """
        Make presence of this node known to others.
        """
        for finger in self.fingerspace.get_all():
            if finger == self.foreign_finger:
                continue
            self.log.info("Announce to %s" % finger)
            announcer = Announcer(self.log, self.local_finger, self.local_keys,
                                  finger)
            announcer.announce()
```

```python
class Announcer(ConnectionHandler):
    """Handler for announcing ourselves to foreign nodes."""
    def __init__(self, log, local_finger, local_keys, foreign_finger):
        """
        :param log: Logger instance to output to.
        :param fingerspace: The FingerSpace instance of this node.
        :param local_finger: The Finger of this node.
        :param local_keys: The CipherWrapper of this node.
        :param foreign_finger: The Finger of the foreign node.
        """
        self.log = log.getChild("announcer@%s" % foreign_finger.ident)
        self.conn = SocketWrapper()
        self.local_finger = local_finger
        self.local_keys = local_keys
        self.foreign_finger = foreign_finger
        self.foreign_key = foreign_finger.get_cipher()

    def announce(self):
        """Send local finger information to a remote node."""
        self.connect()
        try:
            self.send(Protocol.Announce, {'NODE': self.local_finger.all})
        except Exception as exc:
            self.log.error("Announcement Error: %s", exc.message)
        self.close()


class Leaver(ConnectionHandler):
    """Handler for announcing departure to foreign nodes."""
    def __init__(self, log, local_finger, local_keys, foreign_finger):
        """
        :param log: Logger instance to output to.
        :param fingerspace: The FingerSpace instance of this node.
        :param local_finger: The Finger of this node.
        :param local_keys: The CipherWrapper of this node.
        :param foreign_finger: The Finger of the foreign node.
        """
        self.log = log.getChild("announcer@%s" % foreign_finger.ident)
        self.conn = SocketWrapper()
        self.local_finger = local_finger
        self.local_keys = local_keys
        self.foreign_finger = foreign_finger
        self.foreign_key = foreign_finger.get_cipher()

    def leave(self):
        """Send local finger information to a remote node."""
        self.connect()
        try:
            self.send(Protocol.Quit, {'IDENT': self.local_finger.ident})
        except Exception as exc:
            self.log.error("Announcement Error: %s", exc.message)
        self.close()


class IncomingConnection(ConnectionHandler):
    """
    Protocol Handler for communication with foreign nodes.

    The methods of this class define procedures for dealing with connections
    from foreign nodes.
    """
    def __init__(self, log, sock, addr, fingerspace, local_finger, local_keys):
        """
        :param log: Logger instance to output to.
        :param sock: socket object of the incoming connection.
        :param addr: address of the connecting node.
        :param fingerspace: The FingerSpace instance of this node.
```

```
        :param local_finger: The Finger of this node.
        :param local_keys: The CipherWrapper of this node.
        """
        self.log = log.getChild("incoming@%s" % (addr[0],))
        self.conn = SocketWrapper(sock)
        self.fingerspace = fingerspace
        self.local_finger = local_finger
        self.local_keys = local_keys

    def _is_bootstrap_request(self, data):
        """
        Determine if this node is trying to rendezvous.

        Nodes that have not joined the network know of no other node or their
        public key, so they will send their finger information unencrypted to
        a bootstrap node. This attempts to load that data, if it fails then it
        is assumed that it is an encrypted message from an existing node and
        will be handled appropriately.

        :param data: Raw data string received from the foreign node.
        :return: True if this is a bootstrap request, else False.
        """
        try:
            obj = pickle.loads(data)
            assert isinstance(obj, tuple)
            assert len(obj) == 4
            self.foreign_finger = Finger(*obj)
            self.log.info("New node joining network with ID: %s", obj[-1])
            return True
        except Exception:  # pylint: disable=broad-except
            return False

    def _rendezvous(self):
        """
        Accept a new node into the network by sharing our finger table.
        """
        self.log.info("Sending welcome message to %s",
                      self.foreign_finger.ident)
        self.foreign_key = self.foreign_finger.get_cipher()
        parameters = {'NODES': self.fingerspace.export_nodes()}
        self.send(Protocol.Welcome, parameters)
        self.fingerspace.put(*self.foreign_finger.all)

    def handle(self):
        """
        Perform handling of the incoming connection.

        Receives data from the foreign node and deciphers it, this will call
        one of the relevant handlers to deal with the connection based on what
        the message type is.
        """
        data = self.conn.receive()
        if self._is_bootstrap_request(data):
            self._rendezvous()
            return
        foreign, msg_type, parameters = self.unpack(data)
        self._verify_foreign(foreign)
        self._verify_message(msg_type, parameters, None)
        if msg_type == Protocol.Announce:
            self.handle_announcement(parameters)
        if msg_type == Protocol.Quit:
            self.handle_leaver(parameters)
        if msg_type == Protocol.Relay:
            self.handle_relay(parameters)

    def handle_announcement(self, params):
        """Put node information in the FingerSpace"""
```

```
            addr, port, key, ident = params.get('NODE')
            self.log.info("Announcement from %s", ident)
            self.fingerspace.put(addr, port, key, ident)

        def handle_leaver(self, params):
            """Remove a foreign node from network"""
            ident = params.get('IDENT')
            self.log.info('Goodbye to %s', ident)
            self.fingerspace.remove(ident)

        def handle_relay(self, params):
            """Relay package from one node to another"""
            package = params.get('PACKAGE')
            unpacked = self._peel_onion_layer(package)
            if unpacked.get('RECIPIENT') == self.local_finger.ident:
                sender = unpacked.get('SENDER')
                self.fingerspace.put(*sender)
                self.log.info('Message Received from %s', sender[-1])
                print '## Message: %s' % unpacked.get('MESSAGE')
                return
            else:
                addr, port, key, ident = unpacked.get('NEXT')
                self.fingerspace.put(addr, port, key, ident)
                next_finger = self.fingerspace.get(ident)
                self.log.info("Relaying message from %s to %s",
                              self.foreign_finger.ident, next_finger.ident)
                out = MessageHandler(
                    self.log, self.fingerspace, self.local_finger,
                    self.local_keys, next_finger)
                out.connect()
                out.relay(unpacked.get('PACKAGE'))

        def _peel_onion_layer(self, package):
            """Strips a layer from a message package"""
            data = self.local_keys.decrypt(package)
            next_layer = pickle.loads(data)
            return next_layer


class MessageHandler(ConnectionHandler):
    """
    Protocol Handler for outgoing communication with foreign nodes.

    The methods of this class define procedures for dealing with connections
    established locally to transmit to foreign nodes.
    """
    def __init__(self, log, fingerspace, local_finger, local_keys,
                 foreign_finger=None):
        """
        :param log: Logger instance to output to.
        :param fingerspace: The FingerSpace instance of this node.
        :param local_finger: The Finger of this node.
        :param local_keys: The CipherWrapper of this node.
        :param foreign_finger: The Finger of the foreign node.
        """
        self.log = log.getChild("outgoing")
        self.conn = SocketWrapper()
        self.fingerspace = fingerspace
        self.local_finger = local_finger
        self.local_keys = local_keys
        self.foreign_finger = foreign_finger
        if foreign_finger:
            self.foreign_key = foreign_finger.get_cipher()

    def send_message(self, recipient, message):
        """
        Send message.
```

```
        """
        final_pack = self._build_message(recipient, message)
        next_node, params = self._build_onion(recipient, final_pack)
        self.foreign_finger = next_node
        self.foreign_key = next_node.get_cipher()
        self.connect()
        self.send(Protocol.Relay, params)

    def _build_onion(self, recipient, package):
        """
        Construct the onion package
        """
        next_node = recipient
        path = self.fingerspace.get_random_fingers(CFG_PATH_LENGTH)
        try:
            path.remove(recipient)
        except ValueError:
            pass  # We won't route a message to the recipient
        self.log.debug("Path Length %d", len(path))
        _path_msg = " <-- ".join([f.ident for f in path])
        self.log.debug("Path: %s <- %s", recipient.ident, _path_msg)

        for idx, finger in enumerate(path):
            contents = {
                'NEXT': recipient.all if idx == 0 else path[idx-1].all,
                'PACKAGE': package
            }
            cipher = finger.get_cipher()
            package = cipher.encrypt(
                pickle.dumps(contents, CFG_PICKLE_PROTOCOL))
            next_node = finger

        params = {'PACKAGE': package}
        return next_node, params

    def _build_message(self, recipient, message):
        """
        Construct the final message package received by the recipient.

        :param recipient: Finger of the recipient.
        :param message: Textual message for the recipient to receive.
        """
        contents = {
            'MESSAGE': message,
            'RECIPIENT': recipient.ident,
            'SENDER': self.local_finger.all,
        }
        data = pickle.dumps(contents, CFG_PICKLE_PROTOCOL)

        cipher = recipient.get_cipher()
        cryptic_data = cipher.encrypt(data)
        return cryptic_data

    def relay(self, package):
        params = {'PACKAGE': package}
        self.send(Protocol.Relay, params)
```

F.2.6 distrim/ui_cl.py

```

"""
    Command Line User Interface
"""

import sys
import traceback

```

```python
from datetime import datetime as dto

from .node import Node
from .utils.utilities import get_local_ip, format_elapsed


COMMANDS = [
    (('help', 'h'), "Print this message."),
    (('print', 'p'), "Print some information."),
    (('send', 's'), "Send message to node."),
    (('quit', 'q'), "Stop this node and exit."),
]


class CommandLine(object):
    """
    A Command Line Interface (CLI) for controlling an instance of a
    :class:`Node`, including viewing node information and using the network to
    send messages.
    """
    def __init__(self, node_params):
        """
        :param node_params: Dictionary of parameters for :class:`Node`
        """
        self.node = Node(**node_params)
        self.running = False
        self.prompt = "> "
        self._handlers = {
            'help': self.cmd_help,
            'print': self.cmd_print,
            'send': self.cmd_send,
            'quit': self.cmd_quit,
        }

    def enter(self, boot_params):
        """
        Start node and accept commands in a loop.

        :param boot_params: Settings required for starting the node.
        """
        self.node.start(**boot_params)
        self.running = True
        while self.running:
            self.accept_commands()
        self.node.stop()
        sys.exit(0)

    def accept_commands(self):
        """Prompt for command input and execute a command."""
        command = self.get_command()
        try:
            if self.running:
                self.exec_command(command)
        except KeyboardInterrupt:
            print "\n *** KeyboardInterrupt: Command interrupted! ***"
        except Exception:
            print traceback.format_exc()

    def get_command(self):
        """Receive input from the command prompt."""
        command = ''
        try:
            while not command:
                command = raw_input(self.prompt)
            return command
        except KeyboardInterrupt:
            print "\n *** KeyboardInterrupt: Shutting down! ***"
```

```
            self.running = False

    def exec_command(self, command):
        """Parse command and attempt to execute it."""
        cmd, handle, params = self.parse_command(command)
        if not handle:
            print "No such command '%s'." % (cmd,)
            return
        else:
            handle(params)

    def parse_command(self, command):
        """Parse an input command and return a handler."""
        comm, part, params = command.partition(' ')
        for cmd in COMMANDS:
            if comm.lower() in cmd[0]:
                # Return: (long command name, handler)
                return cmd[0][0], self._handlers[cmd[0][0]], params
        return comm, None, None

    # ========== Command Handlers ==========
    def cmd_help(self, params):
        """Input Command: Display Help"""
        print "DistrIM Commands"
        for (longname, shortname), descrption in COMMANDS:
            print "\t%s, \t%s: \t%s" % (longname, shortname, descrption)

    def cmd_print(self, params):
        """Input Command: Print Data to terminal."""
        options = ['crypto-keys', 'fingers', 'node-info', 'node-stats']
        if not params.lower() in options:
            if params:
                print "No such option '%s'" % (params,)
            print "\033[1mPossible options:\033[0m\n  ", '\n  '.join(options)

        params = params.lower()

        if params == 'crypto-keys':
            print "\033[1mNode Keys...\033[0m"
            print self.node.keys.export(text=True, key_type=0)
            print self.node.keys.export(text=True, key_type=1)

        if params == 'fingers':
            print "\033[1mFinger Table...\033[0m"
            fingers = self.node.fingerspace.get_all()
            for fng in fingers:
                print " %s) %s:%d" % (fng.ident, fng.addr, fng.port)

        if params == 'node-info':
            finger = self.node.finger
            print "\033[1mNode Information...\033[0m"
            print "    Hash:", finger.ident
            print " Node IP:", "%s:%d" % (finger.addr, finger.port)
            pubkey = finger.get_cipher().export(text=True)
            print " Pub-Key:", pubkey.replace('\n', '\n' + ' ' * 10)

        if params == 'node-stats':
            print "\033[1mNode Statistics...\033[0m"
            print "Up time:", format_elapsed(dto.now() - self.node.start_time)
            conn = self.node.conn_manager
            print "Succesful Incoming Conns:", conn.count_conn_success
            print "Failed Incoming Conns:", conn.count_conn_failure
            fsi = self.node.fingerspace
            print "Added Keys:", fsi.count_added
            print "Removed Keys:", fsi.count_removed

    def cmd_send(self, params):
```

```python
        """Input Command: Send a message"""
        ident, divider, message = params.partition(" ")
        self.node.send_message(ident, message)

    def cmd_quit(self, params):
        """Input Command: Terminate the node and exit."""
        print "Shutting down..."
        self.running = False


def run_application(args):
    """
    Entry point for the program.

    Initiates the application and fetches any configuration from the program
    arguments.

    :param args: Arguments collected by :module:`argparse`.
    """
    local_ip = get_local_ip()
    if not local_ip:
        print "WARNING: IP Address of this node could not be determined."
        local_ip = raw_input("IP of this node: ")

    params = {'local_ip': local_ip}
    if args.get('listen_on'):
        params['local_port'] = args['listen_on']
    if args.get('logger'):
        params['log_ip'] = args['logger'][0]
        if args['logger'][1]:
            params['log_port'] = args['logger'][1]

    cli = CommandLine(params)

    boot_params = {}
    if args.get('bootstrap'):
        boot_params = {'remote_ip': args['bootstrap'][0]}
        if args['bootstrap'][1]:
            boot_params['remote_port'] = args['bootstrap'][1]

    cli.enter(boot_params)
```

F.2.7 distrim/utils/utilities.py

```python

"""
    Miscelaneous Utility functions.
"""

import socket
import struct
import string
import pickle
# import cPickle as pickle

from random import randint, SystemRandom
from argparse import ArgumentTypeError

from Crypto.PublicKey import RSA
from netifaces import gateways, ifaddresses, AF_INET

from .config import (CFG_SALT_LEN_MIN, CFG_SALT_LEN_MAX, CFG_TIMEOUT,
                     CFG_STRUCT_FMT, CFG_CRYPT_CHUNK_SIZE)
from ..assets.errors import NetInterfaceError, CipherError, SockWrapError


class SocketWrapper(object):
```

```python
    """
    Socket interface for communication with foreign nodes.

    This class wraps around a :class:`socket.socket` object. It provides the
    ability to send and receive packed data, packing it with the length to
    ensure all data is received.

    If the socket is not connected, use the :func:`connect` method to establish
    the connection.
    """
    def __init__(self, sock=None, remote_address=None, timeout=CFG_TIMEOUT):
        """
        Create the wrapper for the sockets.

        Note: You can pass in a socket or an address, if you pass in a
        connected socket, the remote address will be ignored.

        :param sock: the `socket` object. If None, a socket is created using
            the default values.
        :param remote_address: IP and Port of the remote host.
        :param timeout: the timeout value of the socket, how long it will pend
            waiting for a remote response.
        """
        if not sock:
            sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock = sock
        self.remote_address = remote_address
        self.sock.settimeout(timeout)

    def _test_connection(self):
        """Test if connected, raise exception if not."""
        if not self.is_connected():
            raise SockWrapError("Can't use socket, it's not connected.")

    def is_connected(self):
        """
        Determines if the socket is connected or not.
        :return: True if it is, False if it isn't.
        """
        try:
            self.sock.getpeername()
            return True
        except socket.error:
            return False

    def connect(self, remote_address=None):
        """
        Connect the socket to the remote address.

        :param remote_address: IP and Port of the remote host.
        """
        if self.is_connected():
            return

        try:
            if remote_address:
                self.sock.connect(remote_address)
            elif self.remote_address:
                self.sock.connect(self.remote_address)
            else:
                raise SockWrapError("Connect to what? No remote address.")
        except (socket.error, socket.timeout):
            raise SockWrapError("Failure to connect.")

    def close(self):
        """
        Close connection with the foreign node.
```

```
        """
        if not self.is_connected():
            return

        try:
            self.sock.shutdown(socket.SHUT_RDWR)
            self.sock.close()
        except socket.error as exc:
            raise SockWrapError("Error closing socket: %s" % exc.message)

    def receive(self, read_length=1024):
        """
        Receive data from a foreign node via its socket.

        :param read_length: How many bytes to read at a time.
        """
        self._test_connection()
        received_data = ''
        try:
            length = struct.unpack(CFG_STRUCT_FMT, self.sock.recv(4))[0]
            while length > len(received_data):
                stream_out = self.sock.recv(read_length)
                received_data += stream_out
        except (socket.error, socket.timeout):
            raise SockWrapError("Error attempting to receive data.")
        return received_data

    def send(self, data):
        """
        Send data to the foreign node via its socket.

        :param data: The data packet to send.
        """
        self._test_connection()
        length = struct.pack(CFG_STRUCT_FMT, len(data))
        package = length + data
        sent_len = 0
        try:
            while sent_len < len(package):
                sent_len += self.sock.send(package)
        except (socket.error, socket.timeout):
            raise SockWrapError("Error attempting to data data.")


class CipherWrap(object):
    """
    Wrap an RSA Cipher Instance
    """
    def __init__(self, cipher):
        """
        :param cipher: An RSA key, or an RSA instance.
        """
        if isinstance(cipher, basestring):
            try:
                self.rsa_instance = RSA.importKey(cipher)
            except (IndexError, ValueError):
                raise CipherError("Not a valid cipher string.")
        elif isinstance(cipher, RSA._RSAobj):
            self.rsa_instance = cipher
        else:
            raise CipherError("Not a valid cipher.")
        self._has_private = self.rsa_instance.has_private()

    def export(self, text=False, key_type=0):
        """
        Export the RSA key as a string.

```

```
        By default, this just exports a public key in DER format for use in
        fingers.

        If the private key is requested when this instance only has a public
        key, then a :class:`CipherError` is thrown.

        :param text: If True, format the string for humans.
        :param key_type: Key type to export. If 0, public; if 1, private; if 2,
            export public and private key.
        :return: The exported key.
        """
        fmt = 'PEM' if text else 'DER'
        if key_type == 0:
            return self.rsa_instance.publickey().exportKey(format=fmt)
        elif key_type == 1:
            if not self._has_private:
                raise CipherError("Requested non-existant private key.")
            return self.rsa_instance.exportKey(format=fmt)
        elif key_type == 2:
            if not self._has_private:
                raise CipherError("Requested non-existant private key.")
            return (self.rsa_instance.publickey().exportKey(format=fmt),
                    self.rsa_instance.exportKey(format=fmt))
        else:
            raise ValueError("Value for param 'key_type' must be in [0, 1, 2]")

    def encrypt(self, data, split_size=CFG_CRYPT_CHUNK_SIZE):
        """
        Encrypt a packet of data.

        Note that this data must be a string no longer than 128 bytes.

        :param data: The data to encrypt.
        :return: The encrypted data.
        """
        if not isinstance(data, basestring):
            raise CipherError("Can only encrypt string data")

        cryptic = []
        for chunk in split_chunks(data, part_size=split_size):
            cryptic.append(self.rsa_instance.encrypt(chunk, None)[0])
        pseudo = pickle.dumps(cryptic)
        return pseudo

    def decrypt(self, cryptic_data):
        """
        Decrypt a packet of data.

        :param cryptic_data: The encrypted data to decrypt.
        :return: The decrypted data.
        """
        if not self._has_private:
            raise CipherError("Can't decrypt, no private key!")
        pseudo = pickle.loads(cryptic_data)
        data = []
        for chunk in pseudo:
            data.append(self.rsa_instance.decrypt(chunk))
        return ''.join(data)


def split_address(address):
    """
    Transform IPv4 address and port into a `string` and `int` tuple.

    :param address: The string format of the input address.
    :return: tuple of string of the IP or hostname, and port as an int.
    """
```

```python
    parts = address.partition(':')

    if not parts[1]:
        return parts[0], None
    try:
        vals = parts[0], int(parts[2])
        return vals
    except ValueError:
        msg = ("'%s' is not a valid integer in address '%s'"
               % (parts[2], address))
        raise ArgumentTypeError(msg)


def get_local_ip(address_type=AF_INET):
    """
    Determine local IP address of node from its interface IP.

    :param address_type: Any address type from the `AF_*` values in the
        `netifaces` module. Default `AF_INET` for IPv4 addresses.
    """
    default_gateway = gateways().get('default')
    if not default_gateway:
        raise NetInterfaceError("No default gateway found.")

    gateway_ip, interface = default_gateway.get(address_type)
    for addresses in ifaddresses(interface).get(address_type):
        if addresses.get('addr')[:3] == gateway_ip[:3]:
            return addresses.get('addr')


def generate_padding(min_length=CFG_SALT_LEN_MIN, max_length=CFG_SALT_LEN_MAX):
    """
    Create a padding string for use in a cryptographic message

    Generate a random string, of random characters, of a random length for
    padding secure messages.

    :param min_length: Minimum length of the padding.
    :param max_length: Maximum length of the padding.
    :return: The padding.
    """
    uld = string.ascii_letters + string.digits
    length = randint(min_length, max_length)
    pad_list = [SystemRandom().choice(uld) for char in xrange(length)]
    return ''.join(pad_list)


def split_chunks(seq, part_size=128):
    """
    Split a sequence into parts.

    :param seq: The sequence to split.
    :param part_size: Size of the parts.
    :return: Generator function that yields chunks.
    """
    for idx in xrange(0, len(seq), part_size):
        yield seq[idx:idx+part_size]


def format_elapsed(delta):
    """
    Format a :class:`datetime.timedelta` object into a string.
    """
    hours, rem_secs = divmod(delta.seconds, 60*60)
    mins, secs = divmod(rem_secs, 60)
    if delta.days:
        return "%d days, %dh %dm %ds" % (delta.days, hours, mins, secs)
```

```
    else:
        return "%dh %dm %ds" % (hours, mins, secs)
```

F.3 Unit Tests

F.3.1 distrim/unit_tests/test_fingerspace.py

```
# Python testing module:
#     http://docs.python-guide.org/en/latest/writing/tests/

# pylint: disable=protected-access

"""
    Protocol tests, ensures the protocol is handled as we expect.
"""


import unittest
import itertools
import socket

from threading import Thread
from mock import Mock
from Crypto.PublicKey import RSA

from ..utils.utilities import SocketWrapper
from ..fingerspace import (FingerSpace, Finger, generate_hash,
                           finger_type_test, h2i)
from ..assets.errors import FingerSpaceError, FingerError, HashMissmatchError


def a2b(ascii_key):
    """Transforms ASCII public key into a binary one"""
    key = RSA.importKey(ascii_key)
    return key.exportKey(format='DER')


class FunctionTests(unittest.TestCase):
    """
    Checks that the hash function used to assign node IDs is predictable
    """
    def test_hash_ident(self):
        """Test valid data when hasing the ident."""
        addr = '192.168.5.35'
        port = 6050
        pub_key = """-----BEGIN PUBLIC KEY-----
            MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQC+J6zqvL6MHP1Fpn7hmP3xQV/z
            FaZ6p2puVJBaMhwLCh4zTbYlyo/J19Spc9+uqmcFE0z4QEN3AjRdSXvgRH4Qdvvh
            pc9b47cYAUnDd27QIZ/U/FvTcb+Fjhhb3zb+FFvykzGO1YobhaYXQKlnZuFiBq2Z
            JJrG7JW3onqtfHFi4wIDAQAB
            -----END PUBLIC KEY-----"""
        expected = "0f54"

        bin_key = a2b(pub_key)

        result = generate_hash(addr, port, bin_key)
        self.assertEqual(expected, result)

    def test_finger_type_test(self):
        """Tests the :func:`finger_type_test` function."""
        addr = '192.168.5.35'
        port = 6050
        pub_key = """-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQC+J6zqvL6MHP1Fpn7hmP3xQV/z
FaZ6p2puVJBaMhwLCh4zTbYlyo/J19Spc9+uqmcFE0z4QEN3AjRdSXvgRH4Qdvvh
```

```
pc9b47cYAUnDd27QIZ/U/FvTcb+Fjhhb3zb+FFvykzGO1YobhaYXQKlnZuFiBq2Z
JJrG7JW3onqtfHFi4wIDAQAB
-----END PUBLIC KEY-----"""

        bin_key = a2b(pub_key)

        self.assertTrue(finger_type_test(addr, port, bin_key))

        # Types
        with self.assertRaises(FingerError) as exc:
            finger_type_test(192, 6050, bin_key)
        self.assertEqual("ip_address must be a string", exc.exception.message)

        with self.assertRaises(FingerError) as exc:
            finger_type_test('192.168.5.35', '6050', bin_key)
        self.assertEqual("listening_port must be an int",
                         exc.exception.message)

        with self.assertRaises(FingerError) as exc:
            finger_type_test('192.168.5.35', 6050, (bin_key,))
        self.assertEqual("public_key must be a string", exc.exception.message)

        # Values
        with self.assertRaises(FingerError) as exc:
            finger_type_test('localhost', 6050, bin_key)
        self.assertEqual("Invalid IPv4 address: 'localhost'",
                         exc.exception.message)

        with self.assertRaises(FingerError) as exc:
            finger_type_test('192.168.5.35', -6050, bin_key)
        self.assertEqual("invalid port number '-6050'. Must be between"
                         + " 1 and 65535", exc.exception.message)

        with self.assertRaises(FingerError) as exc:
            finger_type_test('192.168.5.35', 6050, pub_key)
        self.assertEqual("public_key must be in binary format",
                         exc.exception.message)

        with self.assertRaises(FingerError) as exc:
            finger_type_test('192.168.5.35', 6050, 'pub_key')
        self.assertTrue(
            exc.exception.message.startswith("public_key is not valid:"))

        private_key = RSA.generate(1024).exportKey(format='DER')
        with self.assertRaises(FingerError) as exc:
            finger_type_test('192.168.5.35', 6050, private_key)
        self.assertTrue(exc.exception.message ==
                        "!!!This is a private key, not public!!!")

    def test_hex_to_int(self):
        """Tests the h2i function"""
        self.assertEquals(h2i('2a'), 42)
        self.assertEquals(h2i('2e'), 46)
        self.assertEquals(h2i('3f2a'), 16170)
        self.assertRaises(ValueError, h2i, 'qa3edsv')
        self.assertRaises(TypeError, h2i, None)


class FingerTests(unittest.TestCase):
    """Test valid creation of a :class:`Finger` object."""
    def test_valid_finger(self):
        """Create fingers with valid data"""
        pubkey = RSA.generate(1024).publickey().exportKey(format='DER')
        obj = Finger('192.168.0.1', 2000, pubkey)
        self.assertIsInstance(obj, Finger)

    def test_invalid_finger(self):
```

```
        """Create fingers with invalid data, expect exceptions"""
        pubkey = RSA.generate(1024).publickey()
        valid_key = pubkey.exportKey(format='DER')
        invalid_key = pubkey.exportKey()

        ips = ['192.168.0.1', None, '', 'localhost', (192, 168, 0, 1)]
        ports = [2000, None, '', '2050', 9999999, -2000]
        keys = [valid_key, None, '', '2050', invalid_key]

        vals = list(itertools.product(ips, ports, keys))

        vals.remove(('192.168.0.1', 2000, valid_key))

        for addr, port, key in vals:
            self.assertRaises(FingerError, Finger, addr, port, key)

    def test_hash_mismatch(self):
        """Tests invalid hash"""
        pubkey = RSA.generate(1024).publickey().exportKey(format='DER')
        self.assertRaises(HashMissmatchError, Finger, '192.168.0.1',
                          2050, pubkey, 'Invalid Hash')

    def test_get_cipher(self):
        """Tests the :func:`get_cipher` method :class:`Finger`"""
        from ..utils.utilities import CipherWrap
        keys = CipherWrap(RSA.generate(1024))
        pubkey = keys.export()
        obj = Finger('192.168.0.1', 2000, pubkey)

        test_data = "This is a beep boop"
        cipher = obj.get_cipher()
        enc_data = cipher.encrypt(test_data)
        self.assertEqual(keys.decrypt(enc_data), test_data)


class FingerSocketTest(unittest.TestCase):
    """Tests the :func:`get_socket` method of :class:`Finger`."""
    def setUp(self):
        """
        Setup to execute before each test.
        """
        from time import sleep
        self.listener = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.listener.bind(('localhost', 0))
        self.listener.listen(0)
        self.listen_thread = Thread(target=self._listen)
        self.listen_thread.start()

        sleep(0.1)  # Momentary pause while the listening socket becomes ready

        pubkey = RSA.generate(1024).publickey().exportKey(format='DER')
        port = self.listener.getsockname()[1]
        self.finger = Finger('127.0.0.1', port, pubkey)

    def _listen(self):
        """
        In a seperate thread, await a connection
        """
        local, address = self.listener.accept()
        self.local = SocketWrapper(local)

    def tearDown(self):
        """
        Cleanup after each test.
        """
        self.listener.shutdown(socket.SHUT_RDWR)
        self.listener.close()
```

```
        self.local.close()

    def test_get_socket(self):
        """
        Receive basic data.
        """
        sock = self.finger.get_socket()
        sock.connect()
        self.listen_thread.join()
        data = '12345678'
        sock.send(data)
        self.assertEqual(self.local.receive(8), data)
        sock.close()


class FingerSpaceTests(unittest.TestCase):
    """Tests the FingerSpace class itself"""
    def setUp(self):
        """Load in test data"""
        import cPickle as pickle
        test_data_path = (__file__.rpartition('/')[0]
                          + '/_testdata_fingerspace.pickle')
        with open(test_data_path) as hand:
            nodes = pickle.load(hand)
        self.local_finger = Finger(*nodes[0])
        self.test_node_list = nodes[1:]
        self.mock_log = Mock()

    def tearDown(self):
        """Test our mock_log each time"""
        self.mock_log.getChild.assert_called_with('fingerspace')

    def test_init_add(self):
        """Initilise and add data"""
        fs1 = FingerSpace(self.mock_log, self.local_finger)
        for addr, port, key in self.test_node_list:
            fs1.put(addr, port, key)

        fs2 = FingerSpace(self.mock_log, self.local_finger)
        for addr, port, key in self.test_node_list:
            ident = generate_hash(addr, port, key)
            fs2.put(addr, port, key, ident)

    def test_add_invalid(self):
        """Tests invalid hash error"""
        invalid_hash = "This is an invalid hash"

        fsi = FingerSpace(self.mock_log, self.local_finger)
        for addr, port, key in self.test_node_list:
            with self.assertRaises(HashMissmatchError):
                fsi.put(addr, port, key, invalid_hash)

    def test_add_valid_duplicate(self):
        """Tests when adding identicle fingers, second is ignored"""
        fsi = FingerSpace(self.mock_log, self.local_finger)
        addr, port, key = self.test_node_list[0]
        self.assertEqual(len(fsi), 0)
        fsi.put(addr, port, key)
        self.assertEqual(len(fsi), 1)
        fsi.put(addr, port, key)
        self.assertEqual(len(fsi), 1)

    def test_add_invalid_duplicate(self):
        """Tests when two different fingers have same hash, logs warning"""
        fsi = FingerSpace(self.mock_log, self.local_finger)
        addr, port, key = self.test_node_list[0]
        bad_finger = Finger(addr, port, key)
```

```
        bad_finger.addr = '0.0.0.0'
        bad_finger.port = 0
        fsi._keyspace[h2i(bad_finger.ident)] = bad_finger
        self.assertEqual(fsi.log.warning.call_count, 0)
        fsi.put(addr, port, key)
        self.assertEqual(fsi.log.warning.call_count, 1)

    def test_add_self(self):
        """Tests that it's considered an error """
        fsi = FingerSpace(self.mock_log, self.local_finger)
        addr, port, key = self.local_finger.values
        fsi.put(addr, port, key)
        self.assertEqual(fsi.log.warning.call_count, 1)

    def test_single_finger(self):
        """Tests adding, getting, and removing a Finger"""
        addr, port, key = self.test_node_list[0]
        finger = Finger(addr, port, key)
        ident = finger.ident
        fsi = FingerSpace(self.mock_log, self.local_finger)

        fsi.put(addr, port, key)
        self.assertTrue(len(fsi._keyspace.keys()) == 1)
        self.assertEqual(finger, fsi._keyspace[h2i(ident)])

        self.assertEqual(fsi.get(ident), finger)

        self.assertTrue(fsi.remove(ident))
        self.assertTrue(len(fsi._keyspace.keys()) == 0)

    def test_empty(self):
        """Tests an empty FingerSpace"""
        fsi = FingerSpace(self.mock_log, self.local_finger)
        self.assertEqual(fsi.get('abcd'), None)
        self.assertFalse(fsi.remove('abcd'))
        self.assertRaises(FingerSpaceError, fsi.get_random_fingers, 1)

    def test_path(self):
        """Test ability for path creation"""
        fsi = FingerSpace(self.mock_log, self.local_finger)
        for addr, port, key in self.test_node_list:
            fsi.put(addr, port, key)

        all_fingers = [x[1] for x in fsi._keyspace.items()]

        lengths = [1, 2, 5, 10, 14]
        for length in lengths:
            path = fsi.get_random_fingers(length)
            self.assertEqual(len(path), length)
            for finger in path:
                self.assertIn(finger, all_fingers)

        self.assertRaises(ValueError, fsi.get_random_fingers, 0)

        path = fsi.get_random_fingers(5000)
        self.assertEqual(len(path), len(self.test_node_list))

    def test_import_and_export(self):
        """Tests importing and exporting values."""
        fs1 = FingerSpace(self.mock_log, self.local_finger)
        for values in self.test_node_list:
            fs1.put(*values)

        expected = [Finger(*node).all for node in self.test_node_list]
        gotten = fs1.export_nodes()
        expected.sort()
        gotten.sort()
```

```python
        self.assertListEqual(gotten, expected)
        fs2 = FingerSpace(self.mock_log, self.local_finger)
        fs2.import_nodes(expected)
        self.assertDictEqual(fs1._keyspace, fs2._keyspace)
        self.assertFalse(self.mock_log.warning.called)

    def test_get_all(self):
        """Test the get_all function"""
        fsi = FingerSpace(self.mock_log, self.local_finger)
        for addr, port, ident in self.test_node_list:
            fsi.put(addr, port, ident)

        fingers = fsi.get_all()
        expected = [Finger(*pars) for pars in self.test_node_list]

        for finger in expected:
            out = fsi.get(finger.ident)
            self.assertIn(out, fingers)
```

F.3.2 distrim/unit_tests/test_protocol.py

```python
# Python testing module:
#     http://docs.python-guide.org/en/latest/writing/tests/

"""
    Protocol tests, ensures the protocol is handled as we expect.
"""


import unittest
from mock import Mock
from itertools import product

import pickle

from ..protocol import (Protocol, ConnectionHandler, IncomingConnection,
                        MessageHandler)
from ..fingerspace import Finger
from ..assets.errors import ProtocolError, ProcedureError
from ..utils.utilities import CipherWrap


class ProtocolTest(unittest.TestCase):
    """Tests the :class:`Protocol` class."""
    def test_protocol_definition(self):
        """
        Sanity test for the class :class:`Protocol`.
        """
        # Combine known attributes in the class
        attrs = [attr for attr in dir(Protocol) if not attr.startswith('_')]
        attrs.remove('ALL')
        attrs.sort()  # Get Protocol message list alphabetically.

        # Create an instance of that class for attribute checking
        proc = Protocol()

        remaining_attributes = [atr for atr in attrs]
        found_attribute_vals = []

        # Test that all attributes are in ALL
        self.assertTrue(len(Protocol.ALL) == len(remaining_attributes))

        for idx, attribute in enumerate(attrs):
            value = proc.__getattribute__(attribute)
            self.assertEqual(Protocol.ALL[idx], value)
            found_attribute_vals.append(value)
            remaining_attributes.remove(attribute)
```

```python

        self.assertTrue(len(remaining_attributes) == 0)

        # Test that attributes are valid for the protocol
        for attribute in found_attribute_vals:
            self.assertEqual(len(attribute), 4)
            self.assertEqual(attribute.upper(), attribute)
            self.assertIn(attribute, Protocol.ALL)


class MockSock(object):
    """Emulates a SocketWrapper object"""
    def __init__(self):
        self.data = ''

    def send(self, data):
        self.data = data

    def receive(self):
        val = self.data
        self.data == ''
        return val


class ConnHandleInit(ConnectionHandler):
    """Extends the abstract class ConnectionHandler with an init method"""
    def __init__(self, local_keys, local_finger, foreign_finger):
        self.local_keys = local_keys
        self.local_finger = local_finger
        self.foreign_finger = foreign_finger
        self.foreign_key = foreign_finger.get_cipher()
        self.log = Mock()


class ConnectionHandlerFuncTests(unittest.TestCase):
    """Test the functions of the abstract ConnectionHandler class"""
    def setUp(self):
        test_data_path = (__file__.rpartition('/')[0]
                          + "/_testdata_protocol.pickle")
        with open(test_data_path) as handle:
            test_data = pickle.load(handle)
        self.nodes = []  # Gives 5 test nodes
        for val in test_data:
            keys = CipherWrap(val['priv'])
            finger = Finger(val['ip'], val['port'], val['pub'])
            self.nodes.append((keys, finger))

    def test_testdata_integrity(self):
        """Ensure our test data is valid"""
        for keys, finger in self.nodes:
            self.assertEqual(keys.export(), finger.key)

    def test_encrypt_decypt(self):
        """Test the encryption and decryption using public-private keys"""
        from itertools import product
        test_data_1 = "Data length 32 repeated 64 times" * 64   # 2048 bytes
        test_data_2 = "Data leng 32 repeated 512 times." * 512   # 16384 bytes
        for idx, (data_a, data_b) in enumerate(
                product(self.nodes, self.nodes), 1):
            node_a = ConnHandleInit(data_a[0], data_a[1], data_b[1])
            node_b = ConnHandleInit(data_b[0], data_b[1], data_a[1])

            cryptic = node_a.foreign_key.encrypt(test_data_1)
            decryptic = node_b.local_keys.decrypt(cryptic)
            self.assertEqual(test_data_1, decryptic)

            cryptic = node_a.foreign_key.encrypt(test_data_2)
```

```
            decryptic = node_b.local_keys.decrypt(cryptic)
            self.assertEqual(test_data_2, decryptic)

    def test_package_unpack(self):
        """Test the packaging and unpackaging of pickled data"""
        test_data_1 = "Data length 32 repeated 64 times" * 64  # 2048 bytes
        test_data_2 = "Data leng 32 repeated 512 times." * 512  # 16384 bytes
        test_dict_1 = {'td': test_data_1}
        test_dict_2 = {'td': test_data_2}
        for idx, (data_a, data_b) in enumerate(
                product(self.nodes, self.nodes), 1):
            print 'Pair #%d' % (idx,)
            node_a = ConnHandleInit(data_a[0], data_a[1], data_b[1])
            node_b = ConnHandleInit(data_b[0], data_b[1], data_a[1])

            cryptic = node_a.package(Protocol.Message, test_dict_1)
            foreign, msg, decryptic = node_b.unpack(cryptic)
            self.assertEqual(Protocol.Message, msg)
            self.assertEqual(test_data_1, decryptic['td'])

            cryptic = node_a.package(Protocol.Message, test_dict_2)
            foreign, msg, decryptic = node_b.unpack(cryptic)
            self.assertEqual(Protocol.Message, msg)
            self.assertEqual(test_data_2, decryptic['td'])

    def test_send_invalid_data(self):
        """Ensure that sending invalid data causes an error."""
        data_a, data_b = self.nodes[0:2]
        con = ConnHandleInit(data_a[0], data_a[1], data_b[1])
        self.assertRaises(ProtocolError, con.send, *("Send", {}))
        self.assertRaises(AttributeError, con.send, *(Protocol.Message,
                                                      "Hello error!"))

    def test_verification_message(self):
        """Ensure that messages are verified properly."""
        data_a, data_b = self.nodes[0:2]
        con = ConnHandleInit(data_a[0], data_a[1], data_b[1])

        self.assertRaises(ProtocolError, con._verify_message, 'Tim', {})
        self.assertRaises(ProcedureError, con._verify_message,
                          Protocol.Ping, {}, Protocol.Pong)
        self.assertRaises(ProtocolError, con._verify_message, Protocol.Ping,
                          {'tim': 'bob'})


class MessageHandlingTests(unittest.TestCase):
    """Test the functions of the MessageHandler class"""
    def setUp(self):
        test_data_path = (__file__.rpartition('/')[0]
                          + "/_testdata_protocol.pickle")
        with open(test_data_path) as handle:
            test_data = pickle.load(handle)
        self.nodes = []  # Gives 5 test nodes
        for val in test_data:
            finger = Finger(val['ip'], val['port'], val['pub'])
            keys = CipherWrap(val['priv'])  # Private Key
            self.nodes.append((finger, keys))  # self.nodes structure

    def test_initial_pack(self):
        """Test direct message passing"""
        for idx, ((fng_a, key_a), (fng_b, key_b)) in enumerate(
                product(self.nodes, self.nodes), 1):
            node_a = MessageHandler(Mock(), None, fng_a, key_a)
            node_b = IncomingConnection(Mock(), None, fng_a.address,
                                        None, fng_b, key_b)

            test_msg = "The quick brown fox jumped over the lazy dog."
```

```python

            cryptic_data = node_a._build_message(fng_b, test_msg)

            unpacked = node_b._peel_onion_layer(cryptic_data)
            self.assertEqual(test_msg, unpacked['MESSAGE'])
```

F.3.3 distrim/utils/unit_tests/test_utilities.py

```python
# Python testing module:
#     http://docs.python-guide.org/en/latest/writing/tests/

"""
    Test cases for utility functions.
"""


import unittest

import socket
import struct
from time import sleep
from threading import Thread
from argparse import ArgumentTypeError
from Crypto.PublicKey import RSA

from ...assets.errors import CipherError, SockWrapError

from ..utilities import (SocketWrapper, CipherWrap, split_address,
                         generate_padding, split_chunks, format_elapsed)


class SocketWrapperListenTest(unittest.TestCase):
    """Tests the :class:`SocketWrapper` class with a listener that creates
    a local socket."""
    def setUp(self):
        """
        Setup to execute before each test.
        """
        self.listener = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.listener.bind(('localhost', 0))
        self.listener.listen(0)
        listen_thread = Thread(target=self._listen)
        listen_thread.start()

        sleep(0.1)  # Momentary pause while the listening socket becomes ready

        self.foreign = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.foreign.connect(('localhost', self.listener.getsockname()[1]))
        listen_thread.join()

    def _listen(self):
        """
        In a seperate thread, await a connection
        """
        self.local, address = self.listener.accept()

    def tearDown(self):
        """
        Cleanup after each test.
        """
        self.foreign.shutdown(socket.SHUT_RDWR)
        self.foreign.close()
        self.listener.shutdown(socket.SHUT_RDWR)
        self.listener.close()
        self.local.shutdown(socket.SHUT_RDWR)
        self.local.close()

```

```python
    def test_receive(self):
        """
        Receive basic data.
        """
        test_str = "Testing String 123. Testing String ABC."
        wrapper = SocketWrapper(sock=self.local, timeout=3)
        data_len = struct.pack(">L", len(test_str))
        package = data_len + test_str
        self.foreign.sendall(package)
        self.assertEqual(wrapper.receive(), test_str)

        # Test big data with 79872 bytes transfered
        big_data = test_str * 2048
        data_len = struct.pack(">L", len(big_data))
        package = data_len + big_data
        self.foreign.sendall(package)
        self.assertEqual(wrapper.receive(), big_data)

    def test_send(self):
        """
        Send basic data
        """
        test_str = "Testing String 123. Testing String ABC."
        wrapper = SocketWrapper(sock=self.local, timeout=3)
        wrapper.send(test_str)
        fetched = self.foreign.recv(1024)
        length = struct.unpack(">L", fetched[:4])[0]
        self.assertEqual(length, len(fetched[4:]))

    def test_send_receive(self):
        """
        Test two sockets sending and receiving
        """
        w_local = SocketWrapper(sock=self.local, timeout=3)
        w_foreign = SocketWrapper(sock=self.foreign, timeout=3)

        test_str = "Testing String 123. Testing String ABC."
        w_foreign.send(test_str)
        w_local.send(test_str)
        self.assertEqual(w_foreign.receive(), test_str)
        self.assertEqual(w_local.receive(), test_str)


class SocketWrapperConnTest(unittest.TestCase):
    """Tests the :class:`SocketWrapper` class with a listener only."""
    def setUp(self):
        """
        Setup to execute before each test.
        """
        self.listener = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.listener.bind(('localhost', 0))
        self.listener.listen(0)
        self.addr = self.listener.getsockname()
        self.listen_thread = Thread(target=self._listen)
        self.listen_thread.start()

    def _listen(self):
        """
        In a seperate thread, await a connection
        """
        self.sock, address = self.listener.accept()

    def tearDown(self):
        """
        Cleanup after each test.
        """
        try:
```

```python
            self.listener.shutdown(socket.SHUT_RDWR)
            self.listener.close()
        except socket.error:
            pass

    def test_init_none(self):
        """Init with nothing."""
        wrap = SocketWrapper()
        self.assertFalse(wrap.is_connected())
        wrap.connect(self.addr)
        self.listen_thread.join()
        self.assertTrue(wrap.is_connected())
        wrap2 = SocketWrapper(self.sock)
        self.assertTrue(wrap2.is_connected())

        test_str = "Testing String 123. Testing String ABC."
        wrap.send(test_str)
        wrap2.send(test_str)
        self.assertEqual(wrap.receive(), test_str)
        self.assertEqual(wrap2.receive(), test_str)

        wrap.close()
        wrap2.close()
        self.assertFalse(wrap.is_connected())
        self.assertFalse(wrap2.is_connected())

    def test_init_sock(self):
        """init with a socket"""
        wrap = SocketWrapper(socket.socket(socket.AF_INET, socket.SOCK_STREAM))
        self.assertFalse(wrap.is_connected())
        wrap.connect(self.addr)
        self.listen_thread.join()
        self.assertTrue(wrap.is_connected())
        wrap.close()

    def test_init_addr(self):
        """init with an address"""
        wrap = SocketWrapper(remote_address=self.addr)
        self.assertFalse(wrap.is_connected())
        wrap.connect()
        wrap.connect()  # Attempting to connect while connected is harmless
        self.listen_thread.join()
        self.assertTrue(wrap.is_connected())
        wrap.close()

    def test_close_error(self):
        """test errors when closing"""
        wrap = SocketWrapper(timeout=1)
        wrap.connect(self.addr)
        self.listen_thread.join()
        self.listener.close()
        with self.assertRaises(SockWrapError) as exc:
            wrap.receive()
        self.assertEqual(exc.exception.message,
                         "Error attempting to receive data.")


class SocketWrapperNoSockTest(unittest.TestCase):
    """Tests the :class:`SocketWrapper` class without external sockets."""
    def test_no_remote_addr(self):
        """Test no remote address"""
        wrap = SocketWrapper()
        with self.assertRaises(SockWrapError) as exc:
            wrap.connect()
        self.assertEqual(exc.exception.message,
                         "Connect to what? No remote address.")

```

```python
    def test_not_connected(self):
        """Test no remote address"""
        wrap = SocketWrapper()
        with self.assertRaises(SockWrapError) as exc:
            wrap.send("Message")
        self.assertEqual(exc.exception.message,
                         "Can't use socket, it's not connected.")

        with self.assertRaises(SockWrapError) as exc:
            wrap.receive()
        self.assertEqual(exc.exception.message,
                         "Can't use socket, it's not connected.")


class TestCipherWrap(unittest.TestCase):
    """Tests the :class:`CipherWrap` class."""
    def test_public(self):
        """Test init and export functions with public key"""
        pk1 = RSA.generate(1024).publickey()  # _RSAobj Instance
        pk2 = pk1.exportKey()  # Text Format
        pk3 = pk1.exportKey(format='DER')  # Binary Format

        cw1 = CipherWrap(pk1)
        cw2 = CipherWrap(pk2)
        cw3 = CipherWrap(pk3)

        # We expect the instance to be created without error
        self.assertIsInstance(cw1, CipherWrap)
        self.assertIsInstance(cw2, CipherWrap)
        self.assertIsInstance(cw3, CipherWrap)

        self.assertEqual(cw1.export(), cw2.export())
        self.assertEqual(cw1.export(), cw3.export())
        self.assertEqual(pk2, cw1.export(text=True))
        self.assertEqual(pk3, cw1.export())
        self.assertFalse(cw1._has_private)
        self.assertFalse(cw2._has_private)
        self.assertFalse(cw3._has_private)

    def test_private(self):
        """Test init and export functions with private key"""
        pk1 = RSA.generate(1024)  # _RSAobj Instance
        pk2 = pk1.exportKey()  # Text Format
        pk3 = pk1.exportKey(format='DER')  # Binary Format

        tup1 = (pk1.publickey().exportKey(format='DER'),
                pk1.exportKey(format='DER'))

        tup2 = (pk1.publickey().exportKey(format='PEM'),
                pk1.exportKey(format='PEM'))

        # We expect the instance to be created without error
        cw1 = CipherWrap(pk1)
        cw2 = CipherWrap(pk2)
        cw3 = CipherWrap(pk3)
        self.assertTrue(cw1._has_private)
        self.assertTrue(cw2._has_private)
        self.assertTrue(cw3._has_private)

        self.assertEqual(cw1.export(), cw2.export())
        self.assertEqual(cw1.export(), cw3.export())
        self.assertEqual(tup1, cw3.export(key_type=2))
        self.assertEqual(tup2, cw3.export(text=True, key_type=2))

    def test_encrypt_decrypt(self):
        """Test the encryption and decryption methods"""
        keys = RSA.generate(1024)  # _RSAobj Instance
```

```python
        private = CipherWrap(keys)
        public = CipherWrap(private.export())

        test_data = "This was a failure test who knows what."
        crypted = public.encrypt(test_data)
        self.assertEqual(private.decrypt(crypted), test_data)

    def test_encrypt_decrypt_big(self):
        """Test the encryption and decryption methods"""
        keys = RSA.generate(1024)  # _RSAobj Instance
        private = CipherWrap(keys)
        public = CipherWrap(private.export())

        test_data = "Data leng 32 repeated 2048 times" * 2048  # 65536 bytes
        crypted = public.encrypt(test_data)
        self.assertEqual(private.decrypt(crypted), test_data)

    def test_invalid_init(self):
        """Test exceptions when creating an instance"""
        with self.assertRaises(CipherError) as exc:
            CipherWrap("NO TIMMY! DON'T DO THAT!")
        self.assertEqual(exc.exception.message, "Not a valid cipher string.")

        with self.assertRaises(CipherError) as exc:
            CipherWrap(('well this is an error',))
        self.assertEqual(exc.exception.message, "Not a valid cipher.")

    def test_invalid_input(self):
        """Test exceptions when using an instance"""
        private = CipherWrap(RSA.generate(1024))
        public = CipherWrap(private.export())

        for _kt in [1, 2]:
            with self.assertRaises(CipherError) as exc:
                public.export(key_type=_kt)
            self.assertEqual(exc.exception.message,
                             "Requested non-existant private key.")

        for _kt in [-1, 3, '0']:
            with self.assertRaises(ValueError) as exc:
                public.export(key_type=_kt)
            self.assertEqual(exc.exception.message,
                             "Value for param 'key_type' must be in [0, 1, 2]")

        with self.assertRaises(CipherError) as exc:
            public.decrypt('anything')
        self.assertEqual(exc.exception.message,
                         "Can't decrypt, no private key!")


class TestAddressSplit(unittest.TestCase):
    """Tests the address split function"""
    def test_valid(self):
        """Expect success"""
        valid_tests = [
            ("localhost", ("localhost", None)),
            ("192.168.0.6", ("192.168.0.6", None)),
            ("localhost:2000", ("localhost", 2000)),
            ("192.168.0.5:3000", ("192.168.0.5", 3000)),
            ("stevie-bob:99999", ("stevie-bob", 99999)),
        ]
        for test_data, expected in valid_tests:
            self.assertEqual(split_address(test_data), expected)

    def test_invalid(self):
        """Expect failure"""
        invalid_tests = [
```

```
            "barry:brought:bacon",
            "192.168.0.1:default",
            "anything::"
            "hostname:"
        ]
        for test_data in invalid_tests:
            self.assertRaises(ArgumentTypeError, split_address, test_data)


class TestPadding(unittest.TestCase):
    """Tests the padding function"""
    def test_padding(self):
        """Test the padding function for correct output"""
        len_min = 64
        len_max = 1024
        test_values = [generate_padding() for cnt in xrange(50)]
        for value in test_values:
            self.assertGreaterEqual(len(value), len_min)
            self.assertLessEqual(len(value), len_max)


class TestSplitChunks(unittest.TestCase):
    """Tests the split chunks function"""
    def test_split(self):
        """Test the values in each part of the split list"""
        test = "This string will be split."
        expected = ["This ", "strin", "g wil", "l be ", "split", "."]
        results = []
        for result in split_chunks(test, 5):
            results.append(result)
        self.assertListEqual(results, expected)

    def test_lengths(self):
        """Ensure the lengths are what we expect."""
        test_list = range(1048641)  # 1024*1024 + 65
        chunks = split_chunks(test_list, 1024)
        self.assertEqual(1025, len(list(chunks)))
        for idx, chunk_gen in enumerate(chunks):
            chunk = list(chunk_gen)
            start = idx * 1024
            if idx == 1024:
                end = 1048640
                self.assertEqual(1024, len(65))
            else:
                end = start + 1023
                self.assertEqual(1024, len(chunk))
            self.assertEqual(start, chunk[0])
            self.assertEqual(end, chunk[-1])

    def test_long_split(self):
        """Split up a long string, put it back together again"""
        test_str = "This could be a very long string indeed." * 5000  # 200000
        out = []
        for idx, chunk in enumerate(split_chunks(test_str)):
            if idx == 1562:
                self.assertEqual(len(chunk), 64)
            else:
                self.assertEqual(len(chunk), 128)
            out.append(chunk)
        reform = ''.join(out)
        self.assertEqual(reform, test_str)


class TestTimeDeltaFormat(unittest.TestCase):
    """Test timedelta utility function :func:`format_elapsed`"""
    def test_format(self):
        """Test it formats correctly"""
```

```python
        from datetime import timedelta

        # days, seconds, microseconds, milliseconds, minutes, hours
        test_cases = [
            ((0, 34, 0, 0, 15, 0), "0h 15m 34s"),
            ((0, 0, 0, 0, 0, 0), "0h 0m 0s"),
            ((1, 0, 0, 0, 0, 0), "1 days, 0h 0m 0s"),
            ((1, 12, 0, 0, 55, 9), "1 days, 9h 55m 12s"),
        ]

        for params, expected in test_cases:
            tdo = timedelta(*params)
            self.assertEqual(expected, format_elapsed(tdo))
```