Invisible Internet Network Protocol (I2NP)

Revision 0.9, 28 August, 2003

http://www.invisiblenet.net/ info@invisiblenet.net

jrandom@invisiblenet.net

Table of Contents

1. Network Overview

Goals

InvisibleNet has formed the Invisible Internet Project (I2P) to support the efforts of those trying to build a more free society by offering them an uncensorable, anonymous, and secure communication system. I2P is a development effort producing a variable latency[1], fully distributed[2], autonomous[3], scalable[4], anonymous[5], resilient[6], and secure[7] network. The goal is to be able to operate successfully in arbitrarily hostile environments – even when an organization with unlimited financial and political resources attacks it. All aspects of the network are open source and available without cost, as this should both assure the people using it that the software does what InvisibleNet says it does, as well as enable others to contribute and improve upon it to make use of newer and better ideas to defeat more aggressive attempts to stifle free speech.

I2P is a peer to peer network that takes advantage of the anonymity and security of mixnets – both free route and mix cascades – the performance, scalability, and resilience of distributed hash tables, and the global interoperability of the Internet. Communication between individuals does not need to expose the location, identity, or contents of those communicating – to each other or to a third party attempting to monitor their activity, even if the third party has unlimited resources dedicated to doing so. Supporting those for whom even the use of this software is deemed illegal is an essential goal which will be achieved through allowing trusted peers, strong cryptography, rotating and expiring physical transport addresses, non traditional communication protocols, and steganographic techniques.

The aim is to provide information theoretic anonymity in addition to the engineering providing computational anonymity, as rotating and expiring physical addresses can by their very nature lose the ability to be tracked after the fact, and trusted routers can be set up to self destruct after a period of time, taking any logs or trace of its installation with it. This works in the same way that a statement such as “Meet me in the garage at 3am tonight and wear a blue hat” does not provide any information as to the location of the target after the scheduled meeting takes place. These rotating and expiring physical transport addresses imply an additional layer of indirection and therefore reduce the latency of the communication so their use should only be used where they make sense, and the same goes for trusted links.

I2P itself isn't an application that people will use in the traditional sense. To understand where I2P fits into the realm of software systems, think of it as a replacement for the Internet Protocol (IP), not as a specific file sharing, instant

1 Variable latency: while every participant has the ability to adjust the response time of their system to meet their anonymity requirements, current models show that I2P will support strong anonymity with up to sub-second latency in a 5+ million node network.
2 Distributed: no centralized point of failure, control, or monitoring.
3 Autonomous: everyone is empowered to operate on the network, and even run parallel networks.
4 Scalable: bandwidth, throughput and latency do not significantly suffer as the network grows.
5 Anonymous: individuals control the disclosure of information about themselves.
6 Resilient: the network can operate and evolve in the face of attacks and failures.
7 Secure: sufficient levels of information integrity and confidentiality are available.

messaging, publishing, or other communication application that runs on top of it. Rather than sending information across a socket or out on a datagram from one fixed location (IP address) to another, you simply send information out on the network to a "destination" (basically a public key), and the I2P software "routers" securely and scalably use whatever means are available to deliver the message, taking into account appropriate anonymity, performance, bandwidth, and reliability constraints, trust metrics, and economic models. An application doesn't need to know where the destination of that message is located, or even how it gets there (and in fact, it can't know either). Inversely, when receiving a message sent over I2P, an application doesn't know where it came from, or even who sent it, unless the sender includes that information. More than that, the machines that route the message from its origin to its final destination have no idea who sent the message or where it is destined, or even if there's a real message involved or if its just random data being propagated for testing purposes.

Assumptions

Some base assumptions are necessary to proceed in specifying the network so that security and anonymity can be defined clearly. The first assumption is that the local computer is secure and can be trusted. This means that there are no key loggers, virii that disclose keys or corrupt the software, or other related faults. The second assumption is that the cryptography used in the network cannot be practically defeated. The quick laundry list of algorithms used includes AES256 in CBC mode, ElGamal with 2048bit keys, DSA with 1024bit keys, and SHA256, in addition to any transport level encryption protocols. The third and final assumption is that the user of the system understands the implications of the security and anonymity constraints and does not violate them. This requires user training so that neither private keys nor identifying data are leaked.

Network Components

The I2P network is made up out of four key abstractions:

- **Routers**: the software responsible for the transparent delivery of messages between destinations. They communicate with other routers to manage tunnels, distribute network information, deliver messages, detect attacks, and detect failures.
- **Destinations**: the unique identifier of an end point in the network to which applications can send messages and receive messages through. Client software uses the Invisible Internet Client Protocol (I2CP) to communicate with a router, providing proof of access to the destination's private key so that the router can securely and anonymously accept messages targeting the destination.
- **Tunnels**: chains of routers collaborating temporarily to pass messages along a fixed path. Messages sent to a tunnel's entry point (gateway) are piped down the network of routers forming the tunnel along with verifying information to assure that the data is altered along the way. In addition, the gateway encrypts instructions to the last router in the tunnel (the end point) telling it how to handle the message. The instructions can specify that a message should be delivered to another router, a `Destination`, another tunnel, or handled locally, in which case the end point consults a local mapping to determine what client to deliver that

message to.

Outbound messages are sent through tunnels built facing away from the source router towards no particular router in the network, along with encrypted instructions specifying where the message should be sent next. This essentially builds the network into a series of disjointed tunnels delivering messages to mobile destinations.

- **Network Database**: a specialized high performance distributed database containing the information necessary to let the network operate effectively. Specifically, this includes the temporary leases of destinations to tunnel gateways and the various pieces of information that routers publish about themselves (their contact addresses, configuration, statistics, etc).

Protocol Scope

The Invisible Internet Network Protocol (I2NP) outlined here is only a part of how an application can send messages over the network. The Invisible Internet Client Protocol (I2CP)[8] defines how client applications written in any language can communicate with the network routers. In addition, various transport protocols define the specifics of how data is passed from one router to another over the network. I2NP does not specify or require any particular transport layer, allowing transport protocols to work over TCP, Polling HTTP, SMTP+POP3/IMAP, UDP, among anything else that can pass data. I2NP merely requires that they:

- Register a unique identifier for use in `RouterAddress` structures consisting of no more than 32 UTF-8 characters.
- Define standard text based options that uniquely define a contact method (for example “hostname” and “port” or “email address”) as usable in the `RouterAddress` structure's set of options.
- Provide a means to reliably deliver a chunk of data, where the contents of any particular chunk is delivered in order. However, different chunks of data do not need to be delivered in order.
- Secure the chunks of data from alteration or disclosure (e.g. encrypt them and use checksums).
- Enable the router to control the transport's bandwidth usage.
- Provide estimates for the latency and bandwidth associated with passing a chunk of data.
- Provide a programmable interface suitable for integration with various routers.

Transports themselves can implement advanced features, such as steganography, constant rate delivery, dummy message delivery, and may even run on top of existing networks, such as mixminion, kazaa, gnunet, and freenet. Transports can even be written to run over I2P itself, accessing it as a client and mixing the message through other routers.

Sandwiched between I2CP and the various I2P transport protocols, I2NP manages the routing and mixing of messages between routers, as well as the selection of what

8 http://wiki.invisiblenet.net/iip-wiki?I2P

transports to use when communicating with a peer for which there are multiple common transports supported.

Threat Model

I2P aims to operate in hostile environments, including against a global active adaptive attacker with infinite resources available to them. This type of attacker can operate internationally in any jurisdiction, actively participate in the network, as well as refocus their efforts to explore potentially exploitive situations. For an analysis of the performance and security of the I2NP, please see the evolving analysis[9].

9 http://wiki.invisiblenet.net/iip-wiki?I2PSecurity

2. Communication Scenarios

Outlined below are the different events and flows that occur within the I2NP. The details of the contents of various messages are located below in section 4, and common I2P data structures (shown like `this`) are located in the data structure spec[10].

Active Scenarios

Router Starts Up

Each and every time the router starts up, it boots the Network Database by contacting known peers and following the process described in the Network Database section below. The first time the router starts up, it must generate its `RouterIdentity` (containing various public keys as well as a `Certificate`) and be provided with seed `RouterInfo` structures with reachable `RouterAddresses` to join the network through.

After booting up the Network Database, it publishes its `RouterInfo` structure into the network database under the SHA256 of its `RouterIdentity`, as well as any `LeaseSet` structures to be placed under the SHA256 of their associated `Destinations`.

If the router's `RouterInfo` does not contain reachable contact addresses (which is the case if the new router choses to run over trusted links only), all attempts to contact the router should be source routed through one of the trusted links published in the `RouterInfo` structure as `RouterSighting` structures.

Router Leaves Network

When a router leaves the network, no activity needs to occur. The various routers with references to the router should detect that it is unreachable and stop using it. The router may want to update the Network Database with a `RouterInfo` structure containing no `RouterAddress` or `RouterSighting` structures, as well as update any associated `RouterAddress` to remove all `Leases`.

Destination Established a Session with a Router

When a `Destination` establishes a session with a router according to the Invisible Internet Client Protocol (I2CP), it provides proof that the client has access to the private key associated with the `Destination`. Based both on the router's configuration and on the configuration properties provided during the session establishment, the router then builds various inbound and outbound tunnels. After these tunnels have been built, the router asks the client to authorize the inbound tunnels by providing the gateways and tunnel identifiers to the client and then waits for the client to provide the appropriate `Lease` structures. Once the router has them, it publishes a `LeaseSet` to the Network Database keyed off the SHA256 of the `Destination`.

Tunnel creation is a three step process. First, the controlling router selects a set of known routers to participate in the tunnel. Next, the router contacts each of those routers with a TunnelCreateMessage telling the router what its next hop should be, among other configuration options, and offering a `Certificate` in exchange for participating in the tunnel. These messages should be sent either through existing

10 http://wiki.invisiblenet.net/iip-wiki?I2P

outbound tunnels or source routed through GarlicMessages and should include a `SourceRouteBlock` on which replies should be sent. After the TunnelCreateMessage is sent to each router, the controlling router waits for a reply from them with a TunnelCreateStatusMessage before moving on to the next step. Prior to sending that TunnelCreateStatusMessage reply, the participating router should make sure it is able to reach the next step – perhaps even establishing a long lasting transport specific connection. Finally, once all replies have been received, the tunnel is complete.

The end points and gateways for both inbound and outbound tunnels receive the `TunnelSessionKey`, which is used for encrypting the `DeliveryInstructions`. All participants in tunnels also receive a `TunnelConfigurationSessionKey` that must accompany any further tunnel configuration message, and they also receive a `TunnelSigningPublicKey` that is used to verify data in the `TunnelVerificationStructure`, while only the gateway receives the `TunnelSigningPrivateKey` for generating that signature.

Router Rebuilds Tunnels

Periodically, routers will want to update both their inbound and outbound tunnels. This may be required if routers participating in the tunnel only agree to a limited time, or it may be done for anonymity and security reasons, altering the overall network traffic flow to the destination as well as providing different routers which an attacker would have to compromise. Updating the tunnel has two or more steps. First, the tunnel contacts the new routers that will participate in the tunnel by sending them each a TunnelCreateMessage and waiting for a TunnelCreateStatusMessage in response. If the gateway of the tunnel is not being replaced, the controlling router needs only to then send a TunnelCreateMessage to the last router in the existing tunnel that will remain as a part of the tunnel, directing it to forward messages to the new series of routers, and then send a TunnelDestroyMessage to the routers no longer serving as a part of the tunnel. However, if the gateway to an inbound tunnel is being replaced, the router should request a new `Lease` from the client and update the Network Database to contain the updated `LeaseSet` prior to sending the TunnelDestroyMessage.

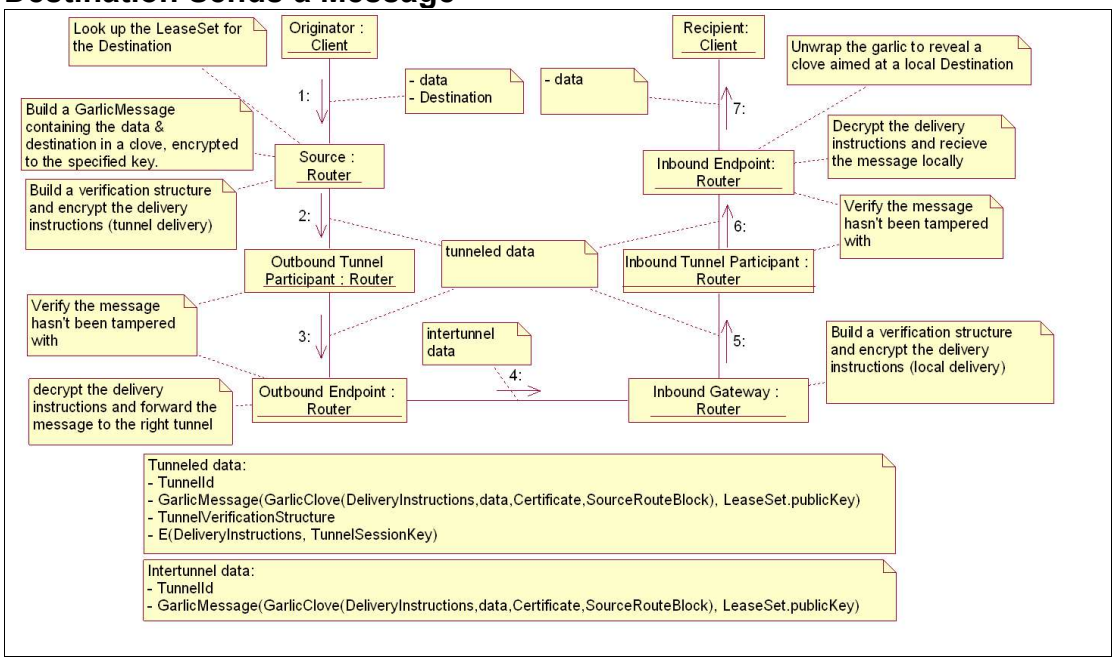
Tunnel Fails

Periodically, the gateway of a tunnel may send a source routed message down the tunnel. If the tunnel is an inbound tunnel, the router receiving it will take note of its reception and consider the tunnel alive and well. If the tunnel is an outbound tunnel (meaning the message was originated by the controlling router), the message will either be wrapped in a source routed message directing the outbound end point to forward the data back to the controlling router, or the message will be directed towards an inbound tunnel for delivery to the controlling router. If a sufficient number of these test messages fail to arrive within a satisfactory time, the controlling router should rebuild the tunnel as described above.

Destination Ends a Session with a Router

Whenever a client ends a session with the router, the router should update the Network Database to replace the old `LeaseSet` with one that does not contain these `Leases`. In addition, the router may chose to destroy the tunnels with TunnelDestroyMessages, though it may keep them around for reuse.

Destination Sends a Message



Whenever a `Destination` wants to send a message to to another `Destination`, it provides its local router with both the `Destination` structure and the raw bytes of the message to be sent. The router then determines where to send it, delivers it through outbound tunnels, instructing the end point to pass it along to the appropriate inbound tunnel, where it is passed along again to that tunnel's end point and made available to the target for reception. To understand fully, each step in the process must be explained in detail.

- First, once the originating router receives the message and the `Destination`, it attempts to find the `LeaseSet` associated with it as stored in the Network Database under the key calculated by SHA256 of the `Destination`.
- The router then builds a GarlicMessage addressed to the SHA256 of the `PublicKey` from the `LeaseSet` with the real data to be delivered. This GarlicMessage contains at least one `GarlicClove` in which there are instructions to deliver the clove's payload to the `Destination`. Additional cloves may be present, and in fact, if the source router desires guaranteed delivery, it will include a clove requesting source route delivery of a DeliveryStatusMessage back to itself. The body of the GarlicMessage with all enclosed `GarlicCloves` is encrypted to the key specified on the `LeaseSet` using the ElGamal+AES256 algorithm described in the data structure spec.
- The router then selects one or more outbound tunnels through which the GarlicMessage will be delivered.
- Then the router selects one or more of those `Lease` structures from the `LeaseSet` and constructs a TunnelMessage along with `DeliveryInstructions` for the outbound tunnel's end point to deliver the GarlicMessage to the inbound tunnel's gateway router.
- The source router then passes the various TunnelMessages down the outbound

tunnel to that tunnel's end point, where the instructions are decrypted, specifying where the message should be delivered.

- At this point, the end point must determine how to contact the router specified in the decrypted `DeliveryInstructions`, perhaps looking up `RouterInfo` or `LeaseSet` structures in the Network Database, and maybe even delaying a requested period of time before passing on the message.
- Once the tunnel end point has the data it needs to contact the inbound tunnel's gateway router, it then attempts to contact it either directly through one of its public `RouterAddress` or source routed through one of its published trusted peers. Over this medium the tunnel end point delivers the GarlicMessage as it was wrapped by the source router, along with the `TunnelId`.
- Once delivered to the inbound tunnel's gateway, the gateway builds a TunnelMessage wrapping the GarlicMessage, encrypting a `DeliveryInstructions` to specify local delivery upon arrival at the tunnel's end point.
- Once the TunnelMessage is passed down to the end point in inbound tunnel, the router opens the `DeliveryInstructions`, notes the request to deliver it locally, and then proceeds to review the contents of the TunnelMessage's payload, which in this case is a GarlicMessage addressed to the SHA256 of a `LeaseSet` that it has published. It then decrypts the payload of the message with ElGamal + AES256.
- After opening up the GarlicMessage, it reviews each of the `GarlicCloves` and processes them each. Cloves with `DeliveryInstructions` addressed to a local `Destination` are delivered to the associated client application, other cloves asking for local processing (e.g. Network Database messages or DeliveryStatusMessages) are processed, and cloves asking for forwarding to other routers are passed off for delivery.

There are several important points of note in this scenario. First, the source router determines how many messages to send, how many outbound tunnels to send them out, how many inbound tunnels to send them to, and how many cloves should include DeliveryStatusMessage responses. The algorithm deciding these choices depends both on the router implementation as well as the `Destination`'s session configuration options specified to balance the bandwidth, latency, reliability, and anonymity constraints. Also, instead of using outbound tunnels to get the message to the inbound tunnel's gateway, the router may decide to source router the message instead. If the message id for a clove has already been processed or its expiration has passed, the clove is dropped.

Passive Scenarios

Router Receives a Request for Tunnel Participation

When a router receives a request to participate in a tunnel via the TunnelCreateMessage, it is given a set of requested configuration options, such as for how long to operate, how many messages to let pass, how many messages or bytes to let through in a time period, whether or not to generate dummy TunnelMessages, and whether or not to reorder TunnelMessages prior to passing them on. In exchange, a `Certificate` is included which may either have value or represent a proof of work so

as to dissuade routers from consuming more than their share of the network's resources. After deciding whether or not to accept the request to participate in the tunnel, the router replies back with the included `SourceRouteBlock` with a TunnelCreateStatusMessage wrapped in a GarlicMessage.

Router Receives a Garlic Message

Whenever a router receives a GarlicMessage, it checks the `DeliveryInstructions` the to see if it is addressed to the local router, either through the SHA256 of the router's `RouterIdentity` or as one of the keys attached to a locally originated `LeaseSet` structure. If it is not local, it forwards the message on to the router whose SHA256 of the `RouterIdentity` matches the address, perhaps by looking for its address in the Network Database.

If it is local, it decrypts the payload of the message with ElGamal + AES256[11] and unwraps the garlic to determine where to send the sub-messages (`GarlicCloves`) contained within, and if sufficient value is included with their `Certificates`, it forwards them on accordingly. These `GarlicCloves` may or may not be unique, and there may only be one of them. Each of them can request forwarding by means of a `DeliveryInstructions` structure. In addition, each clove has its own payload and may include a `SourceRouteBlock` that can be used to send back replies (for instance, if the clove contains a TunnelCreateMessage, the clove's `SourceRouteBlock` will be used to send back the TunnelCreateStatusMessage).

Router Rotates its Addresses

Periodically and after various events occur, such as a denial of service attack or an instruction from the router's administrative interface, the router may decide to abandon particular contact addresses completely. When this happens, the router rebuilds a new `RouterInfo` structure and publishes it to the Network Database keyed off the SHA256 of its `RouterIdentity`. If these new addresses should only be delivered to trusted links, the router sends a TrustedPeerMessage directly to the trusted peer containing the router's new `RouterAddress` structures.

Router Receives a Network Database Message

When a router receives one of the Network Database messages, it follows the procedure specified in the Network Database section below after validating the structures contained in the message.

11 For the structure and algorithm for ElGamal+AES256, see the data structures spec

3. Network Database

The Network Database for I2NP is a distributed lookup and storage system for handling only two data structures – `RouterInfo` and `LeaseSet`. These two structures are keyed into the system under the SHA256 of the associated `RouterIdentity` and `Destination`, respectively. The Network Database itself does not provide anonymity or hide what it stores in any way – it simply aims to provide fast update and retrieval time, as well as strong fault resistance. To achieve these aims, a modified Kademlia[12] system is used.

Kademlia Variations

Except for a few properties outlined here, I2NP's Network Database is an implementation of Kademlia. The differences are:

- A 256 bit key space is used rather than the standard 160 bit one, reflecting I2NP's use of SHA256 instead of Kademlia's SHA1.
- Communication between Kademlia nodes is handled through the I2NP standard technique of sending messages for the various remote procedure calls (RPCs) rather than actually sending UDP or ICMP packets. The associated messages – DatabasePing, DatabaseStore, DatabaseLookup, and DatabaseFindNearest, correlating with the Kademlia ping, store, find_key, and find_node RPCs, respectively – are described with the other I2NP messages below. Also, since I2NP cannot assume bidirectional links, Kademlia replies are in their own messages – DatabaseLookupReply and DatabaseFindNearestReply.
- Data stored in the Network Database is verified prior to being accepted or passed on. The `RouterInfo` and `LeaseSet` structures must both be signed with verifiable signatures and placed under the correct key (SHA256 of the `RouterIdentity` and `Destination`, respectively). In addition, values clobber other values if their version is greater than the one already known, but not if they are the same or an older version, even if the signatures match.
- Data is not necessarily expired after 24 hours, data sources do not need to republish daily, and individual routers do not need to republish local structures hourly. However, routers may republish their structures periodically anyway, both to improve their reliability and as network traffic filler.
- DatabaseFindNearest returns `RouterInfo` structures, not (IP, port number, node ID) triples.

Modified Kademlia Protocol Summary

Key Closeness and Location

The measure of how "close" one key is to another is determined by the exclusive-or of the bits in the key. Each router has a Kademlia "node id" (the SHA256 Hash of the router's `RouterIdentity`) and as data is stored in the network, data is placed on the routers where the node id is the closest to the key. In addition, data may be cached along that path to improve future lookups.

12 http://citeseer.nj.nec.com/maymounkov02kademlia.html

Network Database Initialization

When the router is installed and a new `RouterIdentity` is created, there must be at least one reachable peer to which it sends a DatabaseFindNearest keyed off the SHA256 of its own `RouterIdentity` and continues on, resending DatabaseFindNearest messages to routers in each of the key spaces. In addition, as soon as it is available, it inserts its own `RouterInfo` structure with a DatabaseStore message.

Key Space Management

Each router maintains an ordered list of peers whose node id falls into various ranges of keys (the keyspace), where each range is twice as "far away" from the current node id as the previous. As necessary, each router issues a DatabaseFindNearest request for random keys in those spaces to refresh them and uses an algorithm similar to a least recently used queue to expunge keys as better ones are found.

Key Lookups

Key lookups are recursive requests to routers whose node id is closest to the key, where responses either contain the value or they contain references to peers closer to the key. In addition, once a key is found, the value is stored at the peer that the current router would expect the key to be at if they requested it again. These lookups do not necessarily operate serially and should instead have a small parallelization – sending K requests out at once to the K closest peers to a key, and storing a found result at the K closest peers.

Anonymity

Kademlia's use in I2NP is very specialized, not a general purpose data storage system. Even if an attacker compromised all of the routers a particular router spoke with when looking up a particular key, there is no reason to believe the request was due to anything but a periodic refreshing of the keyspace using random keys. In addition, DatabaseStore messages of `LeaseSet` structures can and should be delivered indirectly – either through source routed GarlicMessages or through outbound tunnels so that compromised nodes cannot determine the originator of that structure.

4. Messages

All messages are serialized to begin with a 1 byte `Integer` specifying their type (correlated with the types below).

Network Database Messages

Message:	0 (DatabasePing)
Description:	This message is sent to a router containing a unique identifier and the SHA256 of the `RouterIdentity` sending the message. If the router is online, it should reply with a DatabasePing containing that same unique identifier (and the SHA256 of its own `RouterIdentity`).
Contents:	`2 byte Integer id, SHA256 Hash of the sending router's RouterIdentity`
Responses:	DatabasePing if the id is new
Notes:	

Message:	1 (DatabaseStore)
Description:	This message is sent directly to routers that should store a particular key/value pair, and hence it contains the key for the database along with the value. The DatabaseFindNearest message determines what routers should receive this message.
Contents:	`SHA256 Hash as the key, 1 byte Integer specifying the type of data, then the actual data`
Responses:	None
Notes:	If the type of data is 0, the data is a `RouterInfo`, or if it is 1, the data is a `LeaseSet`

Message:	2 (DatabaseLookup)
Description:	This message is sent in parallel to routers where the SHA256 of their `RouterIdentity` is closest to the key being looked up, where closeness is defined as the XOR of the keys. The number of routers to whom the message is sent in parallel is a tunable parameter. If any of the DatabaseLookupReply responses contain the key, then the value is returned and no more DatabaseLookups are performed, otherwise further requests are sent out to the routers specified in the various DatabaseLookupReply messages. After finding the value, the router may send a DatabaseStore message to the routers in its routing table closest to the key so as to improve later lookups.
Contents:	`SHA256 Hash as the key, SHA256 Hash of the sending router's RouterIdentity`
Responses:	DatabaseLookupReply
Notes:	

Message:	3 (DatabaseLookupReply)
Description:	The DatabaseLookupReply either has the value requested or a set of `RouterInfo` structures that the router responding would send the DatabaseLookup to.
Contents:	`SHA256 Hash as the key, 1 byte Integer specifying the type of data being returned, then a 1 byte Integer specifying the number of RouterInfo structures follow, then either that many structures or the value of the key.`
Responses:	
Notes:	If the number of `RouterInfo` structures is 0, then the remainder of the reply contains the value stored at the key, and if it is 1, then the type of data being returned is ignored. The values for the type of data match the values specified in the DatabaseStore message.

Message:	4 (DatabaseFindNearest)
Description:	This message is sent in parallel to routers where the SHA256 of the `RouterIdentity` is closest to the key space requested.
Contents:	`SHA256 Hash as the key, followed by the SHA256 Hash of the requesting router's RouterIdentity.`
Responses:	DatabaseFindNearestReply
Notes:	

Message:	5 (DatabaseFindNearestReply)
Description:	This message replies to a DatabaseFindNearest with a set of `RouterInfo` structures closest to the key requested.
Contents:	`SHA256 Hash as the key, followed by a 1 byte Integer specifying how many RouterInfo structures follow, then those structures.`
Responses:	
Notes:	

Tunnel Related Messages

Message:	6 (TunnelCreateMessage)
Description:	Request that a router participate in a tunnel, or update an existing router's participation settings if the `TunnelConfigurationSessionKey` and `TunnelId` matches the existing values.
Contents:	`- 1 byte Integer specifying what type of tunnel participant the router should be (1 = gateway, 2 = endpoint, other #s are anything else)` `- If the type of participant is not 2, the SHA256 Hash of the RouterIdentity acting as the next step in the tunnel follows` `- The TunnelId` `- A 4 byte Integer specifying for how many milliseconds the router should participate in the tunnel, starting from the time this message is received` `- The TunnelConfigurationSessionKey unique to this participant in the tunnel` `- a 4 byte Integer specifying the maximum peak number of messages sent per minute` `- a 4 byte Integer specifying the maximum average number of messages sent per minute` `- a 4 byte Integer specifying the maximum peak number of bytes sent per minute` `- a 4 byte Integer specifying the maximum average number of bytes sent per minute` `- a 1 byte Integer containing various configuration flags. The highest order bit specifies whether the gateway should send dummy messages down the tunnel to meet the maximum average (true = 1), and the next highest order bit specifies whether messages should be reordered before passing them down the tunnel (true = 1). The other bits are not yet defined.` `- the TunnelSigningPublicKey` `- if the type of participant is 1, the TunnelSigningPrivateKey follows` `- if the type of participant is 1 or 2, the TunnelSessionKey follows` `- a Certificate for participating in the tunnel` `- a SourceRouteBlock with which the participant can send back their TunnelCreateStatusMessage reply`
Responses:	TunnelCreateStatusMessage
Notes:	

Message:	7 (TunnelCreateStatusMessage)
Description:	The TunnelCreateStatusMessage is a reply to a TunnelCreateMessage, containing the `TunnelId` and whether or not the router agrees to participate in the tunnel. If it does not agree to participate, it also specifies whether it is being rejected because the router is overloaded, the `TunnelId` is already in use on that router, or the router cannot agree to the tunnel's requirements.
Contents:	`TunnelId followed by a 1 byte Integer specifying the participation status`
Responses:	
Notes:	The status is 0 for successfully created or updated, 1 for failed due to duplicate `TunnelIds` , 2 for failed because the router is overloaded, 3 is failed because the `Certificate` is invalid or insufficient, and other values for unspecified failures.

Message:	8 (TunnelDestroyMessage)
Description:	A TunnelDestroyMessage is sent to a router already participating in a tunnel, specifying both the `TunnelId` to be destroyed as well as the `TunnelConfigurationSessionKey` for that router. After receiving this message, the router should no longer forward any messages along the tunnel and may drop any related data.
Contents:	`TunnelId followed by the TunnelConfigurationSessionKey`
Responses:	TunnelCreateStatusMessage
Notes:	

Message:	9 (TunnelMessage)
Description:	Actual message for delivery down a tunnel
Contents:	`TunnelId followed by a 4 byte Integer specifying the size of the payload, then that many bytes. If the TunnelMessage is being sent to a gateway then that is all that is necessary, otherwise if it is being passed down the tunnel, a TunnelVerificationStructure follows and then the DeliveryInstructions encrypted by the TunnelSessionKey with AES256`
Responses:	
Notes:	The AES256 implementation details are described in the data structures spec, using the first 16 bytes of the SHA256 from the `TunnelId` as the AES initialization vector (IV).

Miscellaneous Messages

Message:	10 (DeliveryStatusMessage)
Description:	Contains an acknowledgment that a message was delivered successfully.
Contents:	`4 byte Integer representing the message id that was received, followed by the Date it was received`
Responses:	
Notes:	May be contained in a `GarlicClove` for automated acknowledgments, but it can be sent manually.

Message:	11 (GarlicMessage)
Description:	Contains an encrypted set of `GarlicCloves` which each in turn can be fully deliverable messages
Contents:	`After decryption, the message contains: 1 byte Integer specifying how many GarlicCloves follow, then those cloves, then a Certificate, then a 4 byte Integer serving as a message id, then the expiration Date for the message, then a 2 byte Integer specifying how many SessionTags follow, then that many SessionTags`
Responses:	
Notes:	The GarlicMessage is encrypted with ElGamal+AES256 as outlined in the data structure spec. If the message ID is a duplicate or the expiration date has passed, the clove is dropped. Also, data contained within the cloves can be any type of message.

Message:	12 (TrustedPeerMessage)
Description:	A message passed directly to a trusted peer to request or respond to a request for authorization to have all messages targeting a particular router sent through the peer.
Contents:	`RouterInfo with private data, followed by a RouterSighting structure`
Responses:	TrustedPeerMessage
Notes:	If the receiving router agrees to act as the trusted peer of the requester, it signs the `RouterSighting` and sends back a TrustedPeerMessage

5.Performance Monitoring and Tuning

To facilitate the development of the router software so as to improve its performance, it is useful to have a mechanism to monitor the performance and operation of the deployed routers without violating the security of any of the routers or destinations in use. Within the `RouterInfo` structure itself, there is a set of mappings that can be populated with up to approximately 64KB of data – it is through this that routers may chose to broadcast some performance and operational data. In addition to allowing routers to chose not to include such information, information contained therein is unreliable and may be maliciously incorrect, so no overall performance of the system can be known except with a probabilistic margin of error.

Over time, as new statistics and metrics become interesting, these can be included in the `RouterInfo` structure and old ones removed. Example statistics and metrics include:

- average number of messages passed per hour
- average size of messages passed per hour
- average latency contacting peers
- total number of known peers
- total number of reachable peers
- average number of inbound tunnels created per destination
- average number of outbound tunnels created per destination
- average length of inbound tunnels
- average length of outbound tunnels
- total number of tunnels participated in
- network database stats
- ping times to network lighthouses [google.com, odci.gov, etc]
- current router bandwidth limits

6.Implementation Details

A significant amount of the operation of the network is left up to individual router implementations – I2NP simply allows them to interoperate cleanly and consistently. The following points outline some specific implementation details explicitly left open to interpretation.

- **Peer and Tunnel Selection Algorithm**

Routers need to choose other routers for a variety of reasons – for tunnel participation, for source routing, for network database queries, and for trusted peers.

The algorithm determining what routers should participate in a tunnel should attempt to minimize the likelihood of an attacker compromising all of the nodes in the tunnel, or of the tunnel failing, so routers may want to keep historical performance information about their peers, periodically check them with source routed messages, and attempt to include a large variety of routers in the set to help minimize the likelihood of an attacker expending a large amount of resources to create an attractive honey pot.

The algorithm determining what routers should be used for source routing should operate along similar lines.

The network database query algorithms are defined by the network database implementation and should optimize the distribution for overall network performance.

Trusted peers should be explicitly chosen, though the duration for which a peer should be trusted should take into consideration the risk of using it, the security afforded by the trusted link, and the likelihood of an attacker compromising the trusted node prior to the trust being revoked.

- **Peer and Destination Routing Table Size**

Each router in the network is in control of how many routers they keep in contact with or maintain historical performance information for. At the minimum, each router should maintain the router information for as many routers as required by the network database (approximately log(number of routers in the network)), but may want to include significantly more than that to improve reliability and to provide more options for the various peer selection algorithms.

- **Network Database Caching and Bandwidth Parameters**

Similar to the routing table algorithm, the network database caching algorithm will want to balance the performance increase of looking up keys on the filesystem versus requesting them from other hosts against the storage space involved. Also, the algorithm used to determine how many peers to send network database messages to in parallel should consider the bandwidth available as well the latency required for this particular message (whether it is a client requested lookup, a periodic refresh, a peer response, or another scenario).

- **Quantity and Style of Garlic Messages**

The use of garlic routed messages leaves open a vast array of options for their delivery, such as whether to include redundant cloves routed through different destinations – improving their reliability, whether to group multiple messages for delivery at once – helping deter traffic analysis, whether to include reply blocks –

allowing recipients to respond, and whether to include delivery status messages – helping to automate the verification of guaranteed message delivery.

- **Performance Monitoring and Publishing Style**

The determination of whether to publish statistics – and even whether those statistics should be completely true, largely true, or complete fabrications – is up to the router implementation. In addition, how each individual router retrieves those statistics and whether they use them in their peer selection algorithms is an entirely implementation specific decision.

- **Certificate Acceptance and Generation Policy**

`Certificates` are attached to a wide variety of messages and structures, and the algorithms, minimums, and generation/selection policies for each can be tuned to address denial of service attacks and perhaps to explore the effectiveness of an economic model for the network.

- **Detecting and Correcting Poor Network Integration**

How frequently and with what mechanism routers send messages out into the network to test peers and tunnels is entirely up to the router implementation.

- **Detecting and Responding to Flooding (DoS and DDoS)**

Routers who receive a large amount of messages from inbound tunnels but are still unable to successfully send test messages through may want to consider building additional tunnels. Routers may also refuse to respond to peers who send excessive network database messages and may even be able to make use of transport level detection techniques to attempt to cut off abusive peers, or at least report the abuse to appropriate clients.