A PROPOSAL FOR DECENTRALIZED ANONYMOUS INTERNET

Martin K. Schröder

Decentralized internet is the ultimate dream in what internet should be like – free, untamed, and completely ruled by the parties that are part of it (peers). This means that nothing on such a network should be ”central” in any way, no single party shall have the power to pull the plug on a computer's connection to the network. No one authority on the network should have a ”list” of all other parties. The routing can be restricted as well and just like with IP packets, the routes can be specified manually through a network of trusted peers.

Goals of the GlobalNet

- Decentralized connectivity. This is a major one. A person should be able to connect to the ”internet” anywhere they go. And they should be able to do this not through an internet backbone but through OTHER PEERS.
- The network should seamlessly integrate with already existing technologies. Users who are unable to access the net though another user wirelessly should be able to access the network using conventional means – for example via a normal internet connection.
- No central authority that issues domain names – all domain names are issued by the network. The network keeps copies of all domain names that are present in it along with their public keys. This can be a database that is copyed between nodes in it's entirety – or only partially. Depending on what type of node it is – you wouldn't want to have a temporary node download the whole database – but it would be good to have more permanent nodes have a longer ”memory” and store much more information.
- The integrity of the information has to be ensured to a 100%. The end user has to know that the pages are original data from the source - even if they get the pages or files from another peer who hosts a mirror.

This is where the beauty of the globalnet shows it's full potential – the network is built on top of an existing network technology but CAN be lowered down to the lower levels as well. However it is not dependent on the lower levels. It is just using them for lack of a better choice. If a direct connection (ie peer-to-peer connection is available, then a node will always use that connection to connect to the network). There should be means of connecting a global net using an ad-hoc wireless network for example. There are research projects at the moment looking at doing just that.

GlobalNet is completely encryped and uses it's own routing protocol to effectively find the peers on the network. No peer can be identified because the network uses onion routing protocol to hide both the source and the destination. ~~In order to reach another node, the network broadcasts a name and receives a response with the first hop to which to send the response packet. The rest is handled by that next node and the subsequent nodes after that.~~ In order to find other nodes on the network, the network uses the distributed hash table to get addresses of the rendezvous points through which the other nodes can be reached. Thus there is no need to know the IP address of the target peer – unless as a matter of pure convenience.

Thus, the network is functioning very much like TOR – but is using UDT as the underlying protocol for transmission. This is because UDT (which is based on UDP) allows things like firewall puncturing and is very fast. The unique features of UDP give the application many features that may not be extensively used right now but may be easily implemented in the future without the need to change transmission layer.

Connections to the final destination are established randomly through any number of peers that the client knows IP address to. All communication is encrypted using a layer of encryption for each peer – thus making it impossible for each peer to snoop on the traffic passing through it.

~~The algorithm for choosing the nodes to route through should be completely random and unidentifiable to ensure the perfect anonymity of globalnet. This may come with a performance hit, but compromises will need to be made with regards to performance vs security.~~ The algorithm is random – but at the moment is only choosing nodes from currenctly connected peers. This should be extended to make it search through a database of peers that it can easily extend by querying other peers about which peers they know of. The main issue with this way to extend is that it can be used to find out which nodes are connected to a node – or at some point have been connected to a node. Whether this is a security risk is a matter of speculation. After all, the limited list returned would contain not only currently connected nodes but also other nodes that the queried node may have knowledge about. So it's not entirely useful for tracing the route of the streams.

MULTIPLE CONTACT POINTS

It should be possible to access globalnet in many different ways. ~~The standard way should be to search for globalnet enabled wireless access points within the area.~~ The standard way is to find another peer to which one can connect. One can then establish connection to other peers from that one peer. One way to avoid revealing other peers through queries is to allow sending a request to the peer you are connected to, telling it to establish a connection randomly through it's own peers and forward data to that connection. The downside is that the complete path directly to the destination will have to be revealed to that peer. This is exactly what is avoided by building the route locally and knowing IP addresses of all the nodes that are part of the link.

Anyone who is connected to the network is also by default relaying traffic for the network. This is quite obvious by the way of how it is designed – but still good to mention. Routing traffic within the network (ie without connecting to any outside hosts that are not nodes) is generally safe. However if one allows connecting to outside hosts then the outside host will think that the connection is originating from you and that can present problems – for you – not for the user that you are relaying for.

Therefore it may be reasonable to have access control list that by default allows certain types of connections to be relayed. Among these being: all peer connections, google, yahoo, any other site that is publicly considered as safe. And also the connections may be allowed only to a handful of different ports. Local connections need to be disabled by default as well (ie connections done to local subnet).

~~If an access point is not available then the software should use any interface and attempt to find a globalnet server within reach from where it is at. If it finds a server then it can connect with that peer. However the whole network may not be reachable at all times and so there has to be a way to enumerate all the peers within the network and share this graph with the rest of the net.~~ This has been changed slightly in that it's now access points that matter, but connectivity with other peers. Other peers can be contacted in any number of ways – the traffic can even be wrapped in ping packets if necessary. But knowing a handful of peers to start with is a good idea. Therefore there can be some ”hub” peers that ”know” a lot of people. Through which one may be able to come in contact with more peers. Just like in the real world, if you know people who are willing to introduce you to others, then you can extend your social network that way. One thing that has changed though is that there will be no central server holding information about all the nodes of the network. There is simply no reason to do that – especially with the use of DHT to store data directly on the network.

PROTOCOL

The protocol consists of three parts:

- Locating target nodes.
- Encrypting messages addressed to these nodes.
- Routing the packets appropriately over the nodes in the network.

There are two ways to locate and connect to peers: one is the obvious one by IP address. The other is by their public key. The public key location works like DNS, but instead of having dedicated servers used for name resolution, this approach is completely decentralized and uses a distributed hash table with SHA1 hash of the public key as ”key” and an IP:PORT record as data.

When a peer node connects to the network, it also stotes it's identity in the DHT. This approach is very versatile, because what this means is that for example the node can ”listen” for incoming connections on a different node through which they will be channeled and by doing so essentially make itself a ”hidden service”. So when somebody knows the IDENTITY of the peer, they can look up which IP address they should connect to – but that IP address will not necessarily have to be the actual node which serves the request.

NOTE: the following paragraphs were written previous to the current idea of operation. So they are more of a hypothetical alternative. I still choose to keep the old data in this document, because it shows how the idea has evolved and which approaches are deemed to be less effective.

Target node location is done in a very similar way that TOR uses to locate it's hidden services. Essentially, each node in the network is a hidden service. But the difference is that an extra confirmation procedure is involved in ”acknowledging” a service. For example you say that ”my name is reallycoolservice.onion” and you want to connect to the network. This means that you can also be accessed by that name and anyone else in the network has to be able to access you by that name. But what if there is already a user who is using or has used that name before?

Duplicate names and impersonation is prevented by associating that name with your public key which is also unique to the name. So you can't just send any public key with a name – it has to be a key that is uniquely associated with that name AND also your private key. So you have a public key and a name and you submit it to the network. If the network has never seen either the name or the public key (which is unique per domain), then it will accept you as a newcomer.

Each node keeps a table of all the domains and public keys that it knows. Since there are no IP addresses and since each node can connect to the network from completely different IP addresses, the only way to identify a node is by encryping a secret with that nodes public key and then sending it to the node which decrypts it and encrypts it with your public key and then sends it back to you.

This means that if two different people choose to use reallycoolservice.onion and submit that to the network then the network will reply with many confirmations as to whether the service is authentic or not. The client then looks at how old each record is and determines the one that is most likely to be the correct one. With time, more and more clients will catch on at the original record and so they will refuse to route any messages if the client is not an authentic one.

All statistics are kept in a public database which can be polled by all clients at any time. This database stores the following data:

- *Client onion address*
- *Client public key*
- *The time of the clients first appearance on this node (times are synced between the sendin and the receiving node as well as several time services in order to be completely sure of the time).*
- *Red flags: client generated errors, did not respond, responded with a different key etc. This data can be used by other clients in order to vet a node inside the network.*

Once the data propagates (which can take a few days) it becomes increasingly more difficult to remove a client from the database – or to use another key. This is where we can draw a distinciton with TOR – while TOR uses a hass of the public key as the domain, we need to be able to have domains that are humanly readable but also without compromising security against impersonation attacks.

A distributed database makes this possible because it acts as a collective certificate authority which stores everybody's certifficates and it becomes extremely hard over time to remove a certifficate from the database. Each node basically confirms with other nodes how old a certifficate is and thus if a certifficate is very old many nodes with return the same creation date for that certifficate and any singular attempts at replacing it with something else will ultimately fail because the algorithm will always replace the certifficates in local database with the ones that are confirmed by the most other nodes.

There can be conflits however when two networks that have the same names meet. Let's assume that we have two nets, A and B that have grown independently and have a common name ”cheesy.onion” which is well known by both networks, but belongs to two completely different people. Once the two nets connect, a peer from either network will be able to poll the peers from the other network for the name cheesy.onion and will get two different public keys. How do we then determine which one is the ”real” onion?

In order to battle this we may need to use a portion of the public key as the server address. (needs to be researched).

HOW SERVICES ARE FOUND

NOTE: this idea is loosely rooted in the idea of using bitcoins to entice routing (and thus stability of your service). However, it is probably better to just use the ordinary scheme where the one who establishes a connection knows about all of it's links. This idea is based on the premise that routing to the service is done out of the control of the sender.

The idea is that the services available on the network are distributed the same way as real world goods are distributed through ”resellers”. A vendor does not have to tell his resellers where the service is coming from. He can simply either buy the service himself or procure it from someone else.

If someone is looking to find where the stuff is coming from, they would usually approach one of the resellers. If the reseller is a weak reseller, they will tell where the stuff comes from and then the attacker can move one step up in the chain. The safety and anonymity of the source is primarily dependent on how much control the source has over it's network of resellers.

Each node will decide which services to ”resell” based on how much traffic each service is getting. So nodes and node owners will be interested in routing traffic to popular services. The more a service is

accessed, the more the intermediate source will ultimately be able to earn.

How service portfolio is decided: should either be decided by the software based on a mathematical formula, or be decided partially by the owner of the node. Best to let the network decide based on number of requests. The more requests a service receives, the more likely it is to remain routed through the network. This will ensure that the most accessed services can retain a reasonable bandwidth. However it would also mean that it will take a considerably longer time to find less known services. And also it would mean that the routes to a well known service will be almost set in stone at each node. This would in turn mean that if the intermediate nodes are compromised one by one, the final location of the hidden service can be uncovered.

NOTE: the idea here is anonymous messages that have public key as their ”destination”. Each node then forwards the message to one of the nodes connected to it which has said that they ”can route to that destination”. The approach is very theoretical and has several practical problems that make this alternative more suitable for things like E-MAIL instead of real time streams. However, one approach would be to use the DHT to store the first node that knows where the data should go. But it was already mentioned earlier and will be mentioned again.

However, if the hidden service maintains a network of first and second line resellers, then it will know when somebody is tring to spy on it and so act accordingly by shedding the resellers and keep itself safe. This can be done by determining through which nodes a hidden service will connect to the network and how it will connect to it. Also, when a hidden service moves about, the network will update the route to it slightly. However, also, if the service is to be entirely anonymous, it should also not have routes that are determined by geographic distance but instead have routes that are sporadically scattered around the world.

NOTE: this approach had in mind that we are using a distributed ”wifi” network as a matrix of nodes. That's why it mentions wifi.

The typical problem with TOR is that all of it's relays need to be public in order for them to be used by the network. While nobody can reasonably control all of these relays, it is reasonable to assume that if their locations and IP:s are known, then they can indeed be monitored for incoming and outgoing connections. This is why a distributed database of knowledge is much better and also the information received upon request for new peers should be a limited number of random nodes.

There are two ways to solve the problem here: either you have groups that have their own local networks which they route through, or you don't have any local networks and simply have one global network. The way TOR works is that it lets the client decide which relays the client wants to be using for his circuits. This creates anonymity because the relays don't keep any routing information at all that may lead an attacker to the source. You simply pick the relays that you want to use, construct a package that can be unraveled at each stop and then send it to the first relay. The relays don't know who you are or where the package is going. Unless all three relays are compromised, it's very difficult to uncover the identity of the source at the last hop. The last hop only receives a package scheduled for delivery by another hop. And has thus no idea whether this other hop is the original source or simply another relay.

NOTE: this approach is the one I ended up using in the example code as well. Simply because it is the one that provides absolute best anonymity because none of the other peers need to know about the final destination. The way it works now is like chained proxies – you connect to one host and instruct them to connect to the next. Then send encrypted data to the next which instructs it to connect to the next

hop in the link chain – until you reach the final destination.

In order to make this work nicely, each client needs to have a decent database of relays that they can send their packets through. In TOR, there is a public database composed of all clients who choose to relay traffic for the network. Something to consider: what if all clients were to act as relays by default? So that if you relay (or ”seed” in bittorrent speak) you get better bandwidth than if you don't ”seed” for the network.

NOTE: the idea of making routing the default is discussed further down. With some obvious restrictions for safety, it can work quite nicely.

Think about how tor works with hidden services: a hidden service announces the three different connection points to the client through the central database. The client can then connect to any one of them and send a message to the hidden service telling it about a relay that the client and the server will be both using to bridge the two of their anonymous connections. The only danger is if one or more of these relays that comprise the circuit to the hidden service are compromized. Then it could mean that the hidden service also can be discovered.

NOTE: here I go into a concept of making services files. It's a little far fetched, but the main idea is that only static pages are served, or some kind of combination of custom scripted language and text to make each ”page” as a sort of file that can be downloaded and viewed. I have not thought of any simply way to create interactive services this way though – so it's mainly a concept. Perhaps one that can be used to build a distributed social network (which is mentioned further into the document).

ARCHITECTURE OF A SERVICE

A service should be like a file. It should not be possible to remove a service as long as there are people seeding it and it should not be possible to uncover the real identities of the original seeders due to the very nature of ”everyone has it”.

This means that when something is uploaded to the service, it needs time to propagate through the network. Content is uploaded through public key encryption. The service uploads it's changes to a node – maybe using git or something similar. The changes are then propagated through the network using git pushes and pulls (over anonymous channels of course).

So I can for instance get a version of a site from my trusted source using a git pull. Then I make a change to the site and push it to the network. Over time the changes are synchronised amongst all the nodes that are acting as mirrors for the site. Everything is of course done using hidden services. This way, even all the updates to the site can be posted using an anonymous circuit to the main address of the hidden service and then further distributed to the local computers. Since each update takes a little time to complete, it's important to make it possible for the user to update only ”on demand”.

Thus a user can download the whole source of the site with all the details, including the database and then run queries on it locally and then once they have made some changes, they can upload the site back to the internet and by doing so ”contribute” their share of the site's content.

The challenge here is to make it interactive and real time. Updates will usually take time, but if data necessary for the most common queries can be searched offline, then running such a site would be a breeze. You could for instance become a publisher and thus get access to publishing to the hidden service. Once you do become a publisher, you can get git usernames and publish stuff to the site. While

everyone else can only download and view. Furthermore, an ordinary user will be able to download the whole site and view it realtime offline.

Question: how do we allow users submit things to the site? This could be loosely implemented using the same methodology (that the user has his own little space that he can upload to the site or to the ”hive”). The only one who can make changes to the permissions though is the one who has the private key for the site. Therefore this kind of sharing could also be used for communication in that messages encrypted with the owners public key and sent to the owner, can be left in the git archive only for the owner to download them at some later point in time.

NOTE: here I made a small entry on the idea of implementing site downloads as bittorrent does to transfer it's files. Basically the site is divided into chunks, hashed, and the chunks can then be downloaded from different peers to reconstruct the complete archive. This approach fails completely though when you have dynamic content or make changes to the site – because it requires that the published data does not change in order to maintain it's integrity. Maybe use different ”snapshots” of the site? But that would not be real time either.. so basically only a theoretical idea that can maybe be used for something else.

What if each page you retreive, you could also store locally and become a mirror for? Then the original site will know that you have a mirror for this bit of information on the site. The site can then hash all bits of information (cache them) and store it all in a table that can then be queried at any time to retreive all the partial and nonpartial mirrors of the site. If you want to mirror some speciffic site on the network, then you would have to explicitly specify this and the whole cache will be transfered to you then. But only the cache – not the actual code.

What you can do is take essentially every URI on the site and once a user accesses it in the browser, you store a cached version of that URI on the site map it to the hash of it's content. So each page will have a hash through which it's cached version can be accessed (basically a hash of the URI) and downloaded from any one of the mirrors.

NOTES ON CACHING

There are many unanswered questions in the approach above. The general idea right now is that some form of distributed hash table can be used to store almost any data directly on the distributed network. But the best approach is probably not to use automatic mirroring at all and simply use the network promarily for transferring data.

I think that an idea there was that it's not good to have a ”central server” for anything – and instead even the information should be mirrored. However, mirroring only works for static content. You can't really mirror an interactive site unless you set up the mirror server yourself. And even then it would present problems such as ”if I post a change to the site, how do I then be sure that my changes reach the site where ”bob” will be able to see them”? Basically impractical for dynamic sites – but good for static content.

Perhaps if one can totally rethink how the data is sent to the server, a distributed solution can be created. The ideal solution of course will be where anyone can serve requests done to the site – but where it is also done in a secure fashion. Anti-tampering can be solved by signing all data that the original node sends. But then we have several other questions there that arise on how to get the data from A to B in a secure fashion and without having B as a static destination.

RETREIVING FROM CACHE

NOTE: this section is kept here primarily as a record of previous ideas. You can skip it if caching is not an interesting topic. Later I will more to "pages" and "packs" and social networks where you "submit" signed content – which are a much better applications of this type of caching approach than for data transmission through an anonymous network.

In the globalnet browser, the cache will be shared amongst many computers. So for example, let's go to our cheesy.onion example. You have the site cheesy.onion – when you access this site for the first time, you also get a number of onion addresses to it's mirrors (ie computers who have it cached). The caching should be done completely anonymously and be entirely automatic. So let's say we at random generate a public key and a private key on Alice. Alice now has a hash that identifies her but it's not tied to her person in any kind of way. It's only used to tell cheesy.onion that alice has a cache of some of the onions pages. Each time Alice visits a cheesy.onion page, her browser saves the page and her id is stored on the cheesy.onion server.

Her ID and public key is also stored on the global database server (and mirrors of it). This ID does not tell much more than Alice has been on the site. She doesn't even need to store her own public key on her own computer because it's enough with the private key and her circuit. Now, if Alice chooses to be a mirror host, Alice also sets up a circuit and uploads her public key and access points to the database server. (note: database server can also be cached. If Bob chooses to host a full mirror of the database server, then Bob gets the whole database and can be used to contact alice should the main server be down.

Mirrors are all randomly chosen and only a handful of them is sent with each request back to the client form the server. The client can then keep an aggregation of mirrors just in case. Or simply hold the latest random copy. This way, an attacker (who also runs as a client) can not at any given time enumerate ALL the mirrors. Neither will he ever know of all the mirrors unless he is told.

Alice should also create an address per session so that her visits on different sites can not be linked together. In this way Alice will have maximum anonymity. So for example, for her communications with cheesy.onion she will have one address and one address only which can then be used to access only the cheesy.onion cache on Alice:s computer. - Through the contact points that Alice advertises with the database server – and that only if Alice chooses to serve as a partial or full mirror of cheesy.onion.

PROPOSED FORMAT

The format of each served page will contain three parts:

- The content
- Digital signature for the content.
- Public key of the site (perhaps unnecessary).
- 10 random mirrors of just this page
- 10 random full mirrors of the entire site.

Since each client can act as both a server and a client, we can have a custom version of apache and sql database that is shipped standard with the software and which supports this type of caching and P2P transmission. An embedded version of firefox would suffice. As well as a standalone apache.

The browser will thus be tightly connected to the server and even sometimes use the servers

functionality to serve pages from cache or to serve pages locally.

On the server, there will also be some pages that will not have support for caching. These will be the pages for contact information or payment for example. Because these types of pages are dynamic, it does not make sense to cache them.

There are many similar caveats that have to be taken into account. Basically a whole new way of thinking has to be applied when writing such globally shared services. In order to make them work, they have to be designed in a certain way that guarantees a balance of authenticity and accessability. It should not be possible to knock the server down – but it also should not be possible to impersonate it and fool the unsuspecting visitors about a nodes authenticity.

Since each page and every file is signed, the user is also guaranteed to get unchanged files even from the mirrors. The digital signature of ever document should be rigurously checked in the browser and checked using a previous public key that has been recorded from the original site. This way even PHP files can be transfered to local host after residing on a mirror and the end user can be absolutely certain that the data is in tact.

Few words about locally hosted sites:
Deep copy, or deep mirror, would be a site that is an exact copy of the original site. Still the mirror does not have the private key of the original site owner so the mirror can not change anything on the site. Doing so will break the signatures and the browser on the client end should throw a big fat warning about the integrity of the code. Still if someone wants to run a deep mirror (which would be created by using for example git) then they should be able to do so – all using php, apache and mysql locally.

THE BROWSER BUNDLE
The browser bundle should contain a custom browser – the way it's already being done with TOR browser. And also it should have a fully functional php web server with mysql for hosting sites. So the new GlobalNet browser will be a browser AND a server at the same time. The extra functionality necessary for signing and providing mirrors for each page can perhaps be implemented as an apache module for the XAMPP server.

There are however still many questions that are still unansvered in this area. Questions like: what happens if the user is already running another web server and a database locally? Should we allow an external service? How can the server be secured so that a malicious site can't run malicious code on all of it's mirrors? Can we completely sandbox the server somehow and isolate it from the rest of the file system.

What the mirroring will do is quite apparent – it will make the first ever torrent like service but for webpages. This means that a ”torrent” is very hard to take down and parts of it are shared with tens and maybe hundreds of people who all can share that piece with yet another person. So the web server and sharing and a custom webbrower make up an interesting approach to the future of web browsing when they are tightly integrated.

ACCESSING NORMAL INTERNET
Normal internet will of course be accessible through the network as well – just like with TOR. Basically an anonymous, array based, proxy service that relays through many different hosts before reaching it's destination.

NOTE: this has already been implemented and we have a socks socket listening which then routes the connection through the anonymous network.

MIXING TECHNOLOGIES

A robust network is one that can connect many different technologies together to produce a one larger network. Imagine if linux could only connect to ther linux computers and not anything else? How would that help linux become more widespread? It wouldn't. So interconnectivity is key. The telecom companies are using old technologies even today to route parts of the internet so as to ensure great connectivity for everyone. A DSL network over the phone nets can be seen as one variant of a dark net. It is encrypted, it is impossible for somebody who has access to the phone lines to snoop on it. And it is fast. People have adopted DSL because it lets them access the internet faster. It is likely that the same reasons will be the reasons they would want to accept access to a new technology such as a darknet.

A darknet has to make the following things possible AND secure:

- Unlimited digital content sharing
- True anonymity with no way to snoop on a darknet server (this can be achieved through the mirror system where the server does not reside anywhere speciffic). Web pages have to be decentralized like files in order to really be anonymous and hard to take down.
- Existing links have to be utilized extensively. Routing has to be done over the internet and over other networks. All possibilities of connection have to be utilized in order for the network to be wide and well connected.

We can go as far as having the owner of the server spawn packets with changes (diffs), signing them and then sending them out to the network. The packets will be accepted by anyone who is interested if the signature is correct. Kind of like patches to a source code tree. While patches CAN be applied in no speciffic order, I think it would still be wize to also implement some kind of patch serial number that increases with each commit to the network so that the complete integrity of the final result is maintained.

Some key concepts to think about is: how will the changes be propagated through the network? GIT is a decentralized version control system – maybe some concepts can be borrowed from there.

IMPLEMENTATION CONSIDERATIONS

The implementation has to be written in a way that can easily be ported to other devices. This is important since a router software would benefit a great deal from being able to run on bare bones hardware or perhaps even be embedded. So a clean C implementation would be best. A core written in C and perhaps gui written in python or something else that is much easier to use for such tasks.

NOTE: The current version is written in a mixture of C and C++. We don't use malloc/free/new/delete.

The implementation should be generic and implement a generic socks interface that can then be used to route any other application. There should be no limit to what can actually be served through the network. GlobalNet is an anonymous transportation layer that randomly scatters circuits over a wide area to achieve anonymity. It is not an application layer and should thus be application indifferent.

USING AN ANALOGY FROM SOCIAL NETWORK GRAPHS

Social network graphs provide a very good platform for understanding the way globalnet can work at connecting people together. A node in the social graph has a number of nodes that it ”trusts” more and

so connects through. The news, or the messages come from outside of the social network and are broadcast to the peers that each node is connected to. When I broadcast a message like this and know that all other nodes will broadcast it as well, I can be pretty sure that the destination will catch the message if it is online. If the destination does not respond, then it's probably not online.

NOTE: This approach is also called ”floodnet” (I believe Gnutella used a flood to identify peers). It is however not very practical due to waste of bandwidth.

I can thus take a message and say to my peers ”broadcast this message” and they will broadcast it to their peers. Each time a peer resends the message, it resends it as it's own, encrypted and signed with it's own key. Once the receipient receives the message and is able to make use of it after decrypting it using their own key, they can reply to the message and send it along the same circuit back to the source. The receipient router obviously knows where the message originated from and so can send the response along the same channel.

Another approach here would be that once the receipient receives the message, he can establish it's own channel to communicate back to the source by polling for the source (asking peers if they know where to find the source). Using the same method of discovery – through broadcasts. Basically you ask all of your peers ”Do you know where key X is?” the peers can reply with a YES, I CAN CHECK or NO. A no reply is generated if the peer has previously tried finding the destination but failed – then they default to saying no for a while (perhaps a set delay like 10 min?). An I CAN CHECK response is the default – when a peer does not know if it has the destination peer inside it's network. The YES response is sent if the peer has previously received a response upon being asked to search for the destination.

This system is highly dynamic because the peer discovery is completely distributed and handled by the network. It would be like having a person's name and then asking in the neighborhood ”Do you know this person or someone who knows this person?” and then the neighborhood may reply yes or no or I can see and then get back to you. The yes answer is not going to last long either – only until the person moves somewhere else into another cluster of the network. If a node answers yes and then upon subsequent connect cannot get a response from the destination then it switches it's state to no.

There is no central directory at all and there is no table of peers – unless of course you want to provide some kind of extra services on top of the net, such as DNS.

The peer discovery process is very expensive and should not be done too often.

The catch is that even the final destination host will reply with a YES, I CAN CHECK or NO answer upon being polled for discovery. He can then even relay the traffic to a different network or serve the request himself. He will answer YES and serve the request – he can sometimes (at random) answer I CAN CHECK and simply wait a small fraction before replying. Or it can answer no if the service has been turned off. It should not be possible to identify whether a host is actually hosting the service themselves or getting information from somewhere else. The host can even send an asking packet futher but simply not replying to any responses that it may get (in fact it should not get any response if the destination is itself – but it could get loops in return through other hosts).

We also need to make sure that two reqeusts can not be pending at the same time. This can be done by assigning a unique hash to the connection and ignoring any subsequent questions that arrive. Depending on which peers a peer is connected to, he will have various degrees of anonymity within the network. You may even be connected directly to the target host and not even know it. This is why it is

also important to have adequate packet filtering that disallows direct access to the services that are intended to be only accessed anonymously. For example a web service served through such a net can run on a local port and accessed only through the tunnel. So any packets that are sent to the destination hash are forwarded directly to this internal port. And responses are sent back.

NOTE: we will have to use rendezvous points here. That is the best and fastest approach. The server will connect through tunnels to a few nodes where it will ”listen” for incoming connections, and then advertise these three nodes globally by storing links to them in the distributed hash table.

Since we know many different nodes and should be able to poll for more peers (random number of peers is returned), we can also establish onion circuits through the network without the need to use a central directory server. Making this network very much like tor. The keys to the different nodes however should be distributed peer to peer and not directly. You would thus normally know of more nodes than the number of nodes that you can connect to. But you will only know their hash addresses and not their IP's. ~~You can then poll a node to check if it is online the same way that you would do to discover a route to a hidden service.~~ Instead of polling it's better to simply store access points for the service in the DHT.

Once a node (let's call the node Max) has received a question from node Alex and sent an answer to another node (either YES or NO), the node Max now will send a random answer to any other node that asks the same question. For example if the node Max receives the same question from another node Molly, it will now just pick either a yes or a no. If the answer is YES (I know public key k) then the other node (Molly) will send YES to Alex as well. Alex now will have two connections that have replied with a YES. So it sends back the reply to Alice (the source) and says YES (I can relay traffic addressed to node with public key X (Max)). Alex then picks (at random) either one of the nodes Molly or Max (direct connection) in order to communicate with Max. Alex does not know which one is the real Max though.

Once a circuit is discovered, it stays like that for a while. The timing will probably need to be experimented with. You could break the circuit and flush the tables of Alex so that alex will once again need to poll it's network for route to Max (presumably now choosing a completely different route). How often you would want to flush is a matter of how dynamic the network should be. Since the route discovery is relatively expensive though it should not be done TOO often. However the load is still distributed and since connections need not be established more often than once every second, the effect of this extra overhead of broadcast should be quite negligible.

STARTING POINTS

- Antinat – a socks library. Can be used as example for the internat socks server.
- Socat – a very flexible socket forwarding tool that has many options for forwarding sockets
- Stunnel – an alternative to the openssl binary.
- UDT – a reliable protocol based on UDP with NAT traversal support.

THE LINK LAYER

The central concept in the software is a ”link”. A link is a connection between the host peer and another peer identified by a public key anywhere in the network. Under the link protocol is the transfer protocol that actually takes care of establishing a link across several different hosts.

A link can be a bridge between a local service and the network of peers or it can be a connection to an external host on the internet. ~~In any case, to the network a link is identified by a hash of a public key~~

~~that is used to send data to that link.~~ A link is now simply an advanced form of ”connection” where you supply a path through the network as argument to connect(). It then establishes an anonymous circuit to the destination.

HOW LINKS WORK

You will have client identities and service identities. A client will be able to create identities and access the network using these identities. There will also be access control on the server granting or denying clients access based on their identities. It will however be impossible to "ban" someone from using the network since identities are self generated.

NOTE: this is quite a lengthy description of the link concept when I was still thinking of ways to implement it. The actual implementation is much simpler. We don't need to use separate identities for each link – we just generate identities on the node level. Although we can generate extra identities in order for example to be able to ”login” on a server (after we have registered that identity with the server).

What if there are two clients accessing the neteork using the same identity? The system will deny access to the new identity and notify the original holder of what has happened. Basically nodes that already have a link established with the identity will know when a request to establish a new link arrives.

A link is an application level concept. Links can be initialised by either peer. When a client connects to another peer on the server port and sends their identity, a link to that peer is created on the server and linked to that connection. So on the application level the user may ask the application whether a link exists to another peer. If a link exists then messages can be sent to that peer. If for some reason the network level (net) cant send the link data over the connection, the application level closes the connection and removes the link.

Since a peer has control over which other peers they connect to the network, they can establish a link over an unlimited number of other peers. Suppose I know IP addrrsses of five servers. I could then message server 1 and tell it to establish a link to server 2. Server 1 sets up a connection bridge - it connects to server 2 using it's own identity and connects the local end of the link with it's end of the link that leads to us. Now I can instruct it to send data to that link. I could now write to this link and instruct server 2 to connect to server 3.

And then I can instruct the last server to establish a connection to a real world host and send and receive data from it. Server 1 should not be able to read anything that I send to server 2 so that data should be encrypted with public key of server 2 (which we already know about from before).

We can now send that server our own temporary public key which it is going to use to reply to us. So instead of server 1 sending it it's own public key, it takes the data from the link it has to us and sends that data instead. *NOTE: this is the data forwarding concept – any DATA packets send to server 1 are forwarded as raw content of these data packets to us. Very simply concept.*

So server 2 does not know that it is us sending the packets. To it it appears that its server 1 making the calls. Whether a peer wants to enable "dialing out" to the public internet from it's host should be a setting that the user can modify. The user can for example only allow access to "trusted" servers.

Steps to expose a local service to the p2p network

localhost:80 ↔ [hash] ↔ | p2p network

- local peer loads the ID for the link (public key) and stores it in it's local table
- when another host asks whether local peer can route traffic to the ID, local peer says YES

- when another peer connects and asks to connect to the ID, local peer starts a new connection to the service and links that connection with the incoming connection from the peer.
- All traffic sent from the other peer will be encrypted with the public key ID
- Local peer decrypts the data using the private key and sends it to the socket connected to the service.
- Local peer reads unencrypted data form the service, encrypts it with the public key of the BEGINNING of the tunnel, then sends it over the encrypted connection to the other peer that forwards it further.

Routing a link

- local peer Alice knows that Bob can route connections to ID
- peer John is connected to Alice and sends a request through the connection asking whether Alice can route to ID.
- Alice replies YES
- John asks alice to establish a link to ID and return it's own link ID for that connection
- Alice asks Bob to establish a link to ID
- Bob connects to the service linked to ID and assigns it a local unique id that will be used by Alice to send and receive stuff to and from this new connection through bob.
- Alice creates it's own ID that uniquely identifies the bridge between Bob and John and sends it to John. Now John can send and receive packets to and from Alice using this ID and they will be sent to Bob.
- Now john can do a LNK_Send(ID) or LNK_Recv(ID) on the new link to Alice and the data will be passed to and from the service that Bob is connected to.

EXAMPLE

NOTE: the example may have changed since I wrote this part. So just check the code to see the latest features.

The example is writte in C++/C and features a very basic implementation of the concept described around here in this document. You will find a zip archive with the program down at the end of the document itself (use ”base64 –decode” to decode it).

Description of the program: the program uses SSL over UDT to communicate (this is like doing SSL over UDP (DTLS) but more reliable since it is using the UDT link library.

Parts of the program:

- Network link library. This C library provides a link abstraction layer that hides all the details of ssl and exposes simple link connect, recv and send functions. Using the code in this library you can connect to peers and send data securely.
- P2P client example. This is the client that features a server and a client in one – although it's more precise to simply call it a ”peer client” since it does not connect to a server – but to peers. All peers in the network are essentially just ”nodes”. When a connection is established to a peer,

the peers can communicate with eachother and send messages through eachother to other peers on the network.

- The message layer. This is the further encrypted message layer used to communicate with the final destination. So essentially you further encrypt the messages that you send so that the peers that you send these messages through can not read the contents.

Usage:
start a peer listening on port 9000 and connect to a number of other peers
gclient –listen 9000 –connect ”peer:9000, peer2:9000”

#
gclient --id publickey.pem

link local port through several peers to a remote host. The program will connect to the first peer, then make a connection from the first peer to the second and then instruct the last peer to connect to google and bridge the connection so that google port can be accessed from localhost.
gclient –tunnel ”localhost:1000>peer1>peer2:8080>peer3:9000>google.com:80”

start a tunnel that connects to a computer at a local networkof a link node. The link hash in this case points to a computer on the same network as the last paramenter. If we want to tunnel to a port on that host, we would naturally write localhost:22 as the last parameter.
gclient –no-listen –tunnel ”2222>[link hash]>192.168.0.2:22”

start a socks proxy on port 8080 that goes through five other hosts chosen at random. The last parameter specifies that we intend to connect out to the internet. We could also specify a public peer there so that we can be sure that the internet is accessible from the last host – otherwise our effort will fail.
gclient –proxy ”8080>rand>rand>rand>rand>rand>internet”

NOTE: we now have an option for a ”debug port” where you can connect with netcat and issue commands. Having a command prompt in the application itself was bulky and inefficient.

Once inside the client:
send <hash> message. # sends a message to the hash peer through the relay circuit. Also establishes a circuit to the peer if such circuit does not exist.
list circuits – lists currently established circuits.

The more peers you connect to, the more reach you will have through the network. By default you can simply connect to a set of ”known nodes” which are published somewhere online. Naturally, in a more advanced client these peers will be saved somewhere and you would probably connect to like 10 or 20 peers.

Also, these peers can be ”gateways” to other parts of the network. For example, you could have a peer called ”darkwader” who relays messages for an unknown number of other peers that connect to it – but who do not connect to the main servers. This would essentially mean that if you have a public key to someone on that dark net, you would be able to reach that peer once you are connected to darthwader node. All the routing is done for you automatically.

THE CHALLENGE OF ANONYMITY

It seems that if one wants to have true anonymity then there are several limitations that are purely practical that make the number of implementations that do implement real anonymity quite limited. The main goal is a simple one – to make sure that neither of the peers that communicate with eachother knows who the other peer that they are communicating with is – BUT they should also be 100% sure that the other peer is the same peer they have been communicating with previously. (that's all they need to know – and this is solved with public key encryption).

The disadvantages of the ”can route?” approach.
The approach where a node asks the peers that it is connected to whether they can route packets to a destination has one main weakness and that is that the question has to be asked in the first place. Since the destination is static, if the destination is blacklisted and the network consists of malicious nodes, then they can track who is trying to access that address and that will essentially lead to the source of the request.

NOTE: this is where we switch to the idea of onion routing.

A much better scenario is if you can construct an intended path for the connection in private where only you know how the connection is structured and where you are in control of which nodes it is built through and where this information is strictly private to you. In this approach you don't need to ask questions. You can just send a packet to the fist node that you have chosen and tell it ”forward this message to the next node after you have decrypted it”. NOTE: in practice we tell it to ”connect to that other node” and then it automatically forwards the data packets that we send to it to that other node.

The main issue is ”how can we guarantee that a newly connected peer can find a large anough number of other peers that it can use to route packets?”. We don't want to use a central server.

One approach that I had in mind was like this: you generate your own key (random) then you use that key as a hash to query the distributed hash table for a handful of hosts (in this case the ”closest ones to that hash”) that can then be used for routing. So that with every new random ID, you will get a random list of fresh peers that you can then connect through.

Proposed Approach

NOTE: this was before the DHT idea so it is still suing a ”tracker”. With DHT we don't need a ”tracker” - only an indexer at best.

One solution that comes to mind would be to use a similar system as bittorrent uses. Essentially to access a hidden service (we can just call it a hidden peer), you would download a ”torrent” file that will have information about the tracking services that the peer is using. A tracker would be simply that database server that keeps track of all the routing nodes in the network. You connect to the tracker and receive x number of routing nodes that you can route through.

This tracking file will be signed with the public key of the peer that you are looking to communicate with – as an assurance that they do in fact approve of the setup in question. The tracker will contain the IP addresses and public keys of all the routing nodes that advertise on it. Now you can connect to these peers and these peers can connect through you.

One question that comes to mind is ”would it be wise then to make everybody route traffic through the network by default?”. It may. But we need to understand how tracking works and what is expected of a

routing node in order to make a reasonable assessment of how safe that would be for either of the peers.

Naturally, every operation that requires either of the peers to reveal any kind of information to any peer on the network is a potential security risk. Since the information about all the routing nodes is publicly available, if routing is enabled by default, it would mean that somebody who knows your ip address (such as your ISP) will easily be able to check if you are running the program by doing a lookup in the public database.

However, we augment this problem by not having one central server but MANY. So you can essentially be using a less known service and then it would be harder to find you in a public database, but nontheless your IP will still be present in a public database associated with the use of the program so it may as well be assumed that you are publicly anouncing ”I USE DARKNET” with all the potential consequences that doing that would mean for you. (although it may not have any consequences at all – it's just a thought).

Also, if everybody is by default routing traffic for other peers then the likelyhood of malicious nodes in the network greatly decreases because the total number of nodes greatly increases, majority of whom are ordinary people just like yourself. So even if everyone is present in a public database and using the routing for all kinds of data then using the network becomes relatively safe. The routing option can also then be disabled by the user explicitly allowing them to be completely invisible.

A key distinction to make here is the difference between using the network to access another hidden service on the network vs using it to access an external site. While routing out into the world can present it's own set of challenges where you as the owner of the routing node will be associated with all the activity that someone else is doing through your proxy, routing to another peer is generally very safe because neither you nor them will know of eachothers exact location and the peers in between are all simply intermediates that only forward data without the knowledge of who is sending it and where it is going.

So allowing everyone to route the traffic for everyone else WITHIN the network is a very positive thing for everybody.

What information does the tracker have?
A tracker has information of all the routing proxies that are currently registered with it. A torrent file will then have a list of trackers that can be used to access the network. As a user, you donwload the torrent file and then have access to a load of trackers who each have lists of many people who can essentially route your traffic for you. The torrent also has public key of the hidden service that you are looking to access.

So when you want to publish a hidden service, you will create a key, pick a handful of trackers and then publish the file to an index site. The index site will keep account of all the hidden services that are available on the network and will make it easy for external parties to find these services (since we don't have DNS). It will also make for an opportunity for users to rate the hidden service offsite and ensure good quality of the torrent file – just like when people rate bittorrent torrents. And all this can be done offsite.

To access the hidden service, you will use one of the trackers to retreive a list of relays and then use these relays to establish an anonymous rendezvous connection through a rendezvous point within the network with the service peer just like TOR does. The system works by the other peer first establishing

a connection through a number of peers to a random relay, then anouncing that they are listening on that relay for incoming connections.

Then you establish a connection to the tracker (also through relays to maintain anonymity) and retreive the IP of the rendezvous points that were chosen by the other peer that you want to talk to. Then you establish connection to that peer through the rendezvous point and you two can now communicate. The final connection goes through a number of relays, to a ”middle man” and then through another number of relays to the final destination.

The globalnet client.
The globalnet client will be a program that will take care of name resolution to all the peers that you want to access. For example, assume you download a torrent for a peer that you want to communicate with. You now add this torrent to your client and can assign a hostname to that torrent. The client can then establish a small local web server (or any other service) on your localhost and update ”hosts” file accordingly on your operating system so that you can now access the hidden service by an arbitrary name that you choose yourself and simply doing so through any web browser.

Your browser will resolve the hostname (which will resolve to localhost in this case) then connect to the globalnet client on the defuault port (80) and issue a ”GET”. The client does not even handle the GET but instead just forwards it to the other end of the established connection and then replies with the response. So to the browser this looks exactly as though you were talking to the service directly. The same will be possible to do with any other service. A hidden service can thus be an SSH service, a VPN port or anything else. All data will be simply routed through the network completely transparently to the client application.

NOTE: we now have socks support in the server so with browser socks support any port or service can be accessed through the browser the usual way without making any changes to the hosts file.

A tracker can be any peer in the network. You connect to a tracker the same way as you would connect to a peer and use messages to retreive tracked relays. You can also publish yourself on the tracker's relay list also using messages that you send to that peer. So tracking is part of every client but runs as a side function to it.

Tracker protocol:

- TRACK_PUBLISH publish ourselves to the tracker. The client needs to have relay mode enabled before doing this.
- TRACK_UNPUBLISH unregister from the tracker.
- TRACK_GET_RELAYS returns list of all relays currently tracked by the tracker as a TRACK_RELAY_LIST message.

Relaty protocol:

- RELAY_CONNECT [protocol] [host:port] – connect the current connection to another host and relay data inbetween. Protocol is REL_PROTO_TCP, REL_RPOTO_UDP, or REL_PROTO_INTERNAL. This is the type of connection that is established to the target host. The relay replies with RELAY_CONNECT_OK once the connection has been established.
- RELAY_DATA [data] sends a relay data packet. Used to communicate with the other end of the relay.
- RELAY_DISCONNECT disconnect data output from the relay. Can be sent by either peer to

signal a disconnection from the relay. Note: not used because we simply watch the underlying connection for disconnects.

Each connection can have only one relay destination. If you want to establish a new connection to another host through a relay, you will normally just connect as normal and send the RELAY_CONNECT command to the peer. RELAY_DATA is automatically forwarded to the other end of the established connection.

Example:

- Connection *con is connected to peer George. We want to connect this connection now to Bob through George. So we issue RELAY_CONNECT REL_PROTO_INTERNAL Bob. George established another connection to Bob, but does not do a handshake with Bob, but simply bridges the two connections for us. Any RELAY_DATA packets sent by us to George will be automatically forwarded to Bob so now we can do handshake with Bob and establish an encrypted connection.
- We create a new connection Connection *bob that will lead to Bob and associate it with our previous connection to George but with CON_FLAG_RELAY set so that the underlying logic knows that it should wrap all data as RELAY_DATA packet and send it to *con instead of sending it to a UDP socket.
- The connection issues a handshake and waits until the link is established.
- We can now do CON_sendPacket(bob) to send data directly to bob, but Bob will not know it is us that are sending the packet – to Bob it will look as though it's George that has access to it.
- The data will be encrypted with Bob's public key, then encrypted with George's public key and sent to George, where it is decrypted and sent to Bob, where it is decrypted and consumed by Bob.

Now we can use Bob to establish a connection to the internet. We can for example send REL_RPOTO_TCP (note: we might want to filter the allowed hosts that we allow others to connect to through our relay) to establish a connection to another computer. The outboud connections will probably need to have a rather rigorous set of rules that prevent security holes that come with allowing them. The cool thing is that now we can bypass all firewalls using P2P connections and establish new connections to local hosts or outside hosts from anywhere where we can find a relay.

CERTIFFICATES SIGNING

A hidden service on global net can have an optional signed certifficate which is signed by the main authority (umm... someone who everyone agrees should be allowed to sign certifficates?). We can then approve of their validity and if the certifficate is signed then a user can easily verify that it is valid and that they are in fact talking to the correct peer.

PEGGING CONNECTIONS

Pegging means essentially stacking several connection objects on top of eachother. When two connections are pegged they will be using eachother's virtual functions to create intended logical behavior. For example when two PEER connections are pegged and you call CONNECT on the first one, it will check if it is pegged and if it is, it will use sendCommand(RELAY_CONNECT) instead of connecting directly – which is useful for establishing chained connections.

After this function call, all packets written to source are packaged as CMD_DATA payload and sent to destination. On the destination they are unpacked and forwarded further.

DISTRIBUTED HASH TABLE

To locate hidden services one can use a distributed hash table to find the rendezvous points that the hidden services connect to. So a hidden service publishes a request on the network saying that it's public key hash is using the following rendezvous points through which it can be reached. Then this request is stored thougth the simple ”less than” approach and ends up on one of the nodes in the network. We do need to check regularly whether the key is still there just to make sure that others can reach us but that can be solved by keeping several copies of the key (for example 3?) around the node that stores the main record.

Each request into the network is done through a random relay. So for example a hidden service can pick 2 random hosts to use in order to connect to the third one. The third one is where it will execute it's request to store the key value pair. It will then be the third node that will find the target bucket for the key and store the key there. In the same way we can connect to a rendezvous point and publish an ”access key” there meaning that another node can send information to us directly over that node.

DHT class operates through network. Through the network it can create a link to a random host and publish it's data there. To do this it does the following:

- NET_getRandomLink(net) – returns a 3 hop link to a random host based on the nodes that the current node is currently connected to. Sending data to the link is the same as sending data to a connection to the last node in that link. Although we don't know which node that is.
- Send a DHT_STORE operation with [160bit key] and an arbitrary array of data. (maximum of say 4096 byte.
- The target node then searches it's connections for the node ID that is closest to the DHT key that was provided to it. It then forwards the operation to that node.
- The last node picks three other nodes that it itself is connected to who are closest to the target key and stores three copies of the key there as well.
- We now have a robust hash table to store any value that we want to make accessbile to the whole network.
- The store operation can be repeated with some intervals to ensure that the data always remains published on the network.

To retreive the stored value, we do the following:

- Create a random link to a random node. Then, from that node:
- Construct a query DHT_RETR [key] and send it to the 3 node that have the ”closest” id to that key.
- Monitor response of the nodes and once a response is received, forward it to the node that asked for it. If no response is received from the link then we can assume that the key was not found.
- Otherwise we should get a DHT_DATA packet with the corresponding data. (we should also probably tag the requests with some unique id and then map it with response so that we know which response corresponds to which request.

How to publish a resource.

A resource is essentially a stream. When a user connects to the resource from somewhere else, they can either read the stream or write to it – just like a socket. We can send a request and get a response.

To publish a resource we need two things:

- A resource handler. This can be a function within the code or a socket. We set up a stream class and then simply let the link that we establish read and write this stream.

- A resource ID – this is a SHA1 hash of the resource.

If the resource is a host:port combination then we establish a new connection each time that a new client is looking to access the resource.

Example: simple resource.
Let's publish a file resource. To do this we set up a structure with callbacks that are used to read and write to and from the file. Then we assign an ID to this structure, establish three connections to three random hosts and publish these hosts in the DHT using the resource ID. People can then access the resource using that id. The ID is also a hash of the public key of the resource which will be used to transfer data.

When somebody wants to access the resource, they will query the DHT for the descriptor. If the descriptor is found, then they will pick one of the three nodes which are access points to the resource and connect to them. It will then issue a RES_CONNECT [hash] on the connection. And all CMD_DATA now sent to the connection will arrive on the other end of the pipe.

NOTE: this is not really implemented at the moment so it will probably change once the actual practical implementation is developed.

The front point on the rendezvous point is a shared one though. So through that data connection (when we can talk to the server) we should set up a pseudo connection before handing over the control to the connecting client. We can do so by issuing a RENZV_CONNECT [source id] request to the other end of the connection. The other computer will then create a unique ID for us and send it back which we can then use to communicate with our own instance of the listening socket on the other host. On the other end we have a Listen loop that waits for a connection request. When it accepts a connection, it sends a data packet with an id of the new session and we then use that id to encrypt the traffic we receive on the rendezvous point from the actual client. So the listening socket or the hidden service handler has to be copied for each new client that connects to us.

The service responds with a RENZV_CONNECT_OK [source id] which puts the connection into active state and we can now send RENZV_DATA to the other end of connection. Each such packet contains a destination ID that we were given by the server which is unique for our session. This information is mainly used for routing packets and making sure that they are forwarded to the correct socket on the final node.

ARCHITECTURE
Every connection that goes through the router has it's own set of relays that are randomly chosen by the router. When a listening port is present that tunnels connections to another port on a remote machine then we can connect to that port and we will always get a separate circuit to the remote host that is not shared with other connections. It may be beneficial to look into how a shared connection can be implemented to save the limited number of available ports.

All encryption for the intermediate hosts and relays is done on the client. This is implemented by nesting conneciton objects upon eachother using the function ~~CON_bridge()~~ peg() of the Connection object. This configures each connection object to communicate with the underlying connection object and send data through it. The data then gets sent to the bottom link, decrypted once there and the data of the next link is sent through a raw socket to the next link where it is decrypted again until it reaches the destination.

We must therefore model each link so that we can either do encryption or have an unencrypted connection directly to a normal machine that is not using our protocol. This is done by setting the ”proto” parameter passed to CON_init(). This sets the protocol of the connection link.

For example, when we need to establish a connection to another peer and then connect to a TCP port on another host, then we can create a connection object for the peer using REL_PROTO_INTERNAL_CLIENT. This tells the code to initialize a client connection. We then call CON_connect(). This call connects to the host passed to it and initializes communication. Since we have it set to internal protocol, it will intialize SSL and enable sending of all the RPC commands.

We can now create a connection with REL_PROTO_TCP_CLIENT, bridge it to the first connection and the call CON_connect() on this object. This will now use the RPC calls of the underlying connection to establish a new connection on the other peer leading to the host that we want to connect to. Once the peer has connected to the host it will send back and RPC message RELAY_CONNECT_OK and this connection object will then change it's state to CON_STATE_ESTABLISHED – which will allow us to send data over it as though it was a local TCP connection.

CONNECTIONS

The code is structured so that the same objects can be used on both the server and the client to represent both local and remote connections. Here is how it works.

Each connection object has _input connection object and and _output connection object. In addition to that it also may have a private input output which it will be using to send and receive data. So that for example a TCP socket will have as its output a real TCP socket which will be connected to a remote server. Now when we do a send() on the TCP connection, it will send the data on the next ”update” to the tcp connection and also read data from there and store it in it's internal buffer for us to read later.

A PEER connection has as it's output an SSL connection which in turn has as it's output a UDT connection. This means that PEER connection encodes it's own packets, writes them using send() to the SSL connection – which in turn mangles the packets into it's own format – which is then sent using send() to the UDT connection which in turn writes it to it's internal socket. So if we want to nest two PEER connections, we need to set the _input of the second to the link of the SSL connection of the first and also set the _output of the SSL to the second PEER conneciton. Like this:

PEER2 → SSL → PEER1 → SSL → UDT → INTERNET

This would mean that we will encode data into P2P packets, encrypt them with SSL then package them again as p2p packets destined to another host and then encrypt and send them to the internet.

Why double encryption? Because this setup represents a link that goes through PEER1 and into the internet and then to PEER2. Or more precisely, it is a local data structure to represent a physical anonymous link that looks like this:

WE → PEER1 → PEER2

The data gets first encrypted with PEER2 keys, then with PEER1 keys, then sent to PEER1 where it will decrypt the data and send it to PEER2 who will be able to see our original data.

On PEER1 we have this setup:

WE ← BRIDGE → PEER2 (and on PEER2 we simply have a socket to PEER1 because PEER2 thinks that it is only communicating with PEER1 and has no idea about US)

BRIDGE _input is connected to the _input of our socket on PEER1 and it's output is connected to the _input of the PEER2 socket. What BRIDGE does is simply read data from it's _input and sends it to it's _output and vice versa.

BRIDGE is set up when we send a RELAY_CONNECT packet to the other PEER. We can then set the url to ”tcp:otherhost:port” or ”peer:otherhost:port”. This will instruct the PEER packet handling routine on PEER1 to create a new connection to PEER2 and create a BRIDGE structure which it will fill with input and output.

Some more cases:
When a TCP node is nested on top of a PEER node, it means that the TCP connection has to be done from PEER. This is accomplished by locally calling sendCommand() on the _output pointer – which points to peer::sendCommand – which sends a command packet to the remote host.

When one PEER node is nested on another PEER node it means that the connection to the first PEER node in this data structure should be made from the second. When we call ”connect” on the first link in the chain, it should call sendCommand of the _output node and that in turn will establish a new connection from the peer that it is connected to to the second peer. Then when we call send() on the first PEER, it will translate into DATA encrypted with key for that peer and then sent to send() of the second node – which will also package it as data and then send it to the network. When it arrives on the first peer node, it will decode data, see that it is indeed a DATA package and send it futher to it's _input pointer – which in this case now will be a BRIDGE that will be connected to another peer. So the decrypted DATA – which itself is a DATA packet is now finally sent to the last peer that etither discards it or forwards it to the final host which can be a TCP connection or another node.

It is important to note that if we call sendCommand on one node which is connected to another node, it should be converted to DATA.

OPTIMIZING FOR EFFICIENCY
The program has to meet the following criteria:

- Connections should be fast.
- Resolving a host should be fast.
- Establishing many little tcp connections over a short period of time should be quick (typical of http).
- Transfers should be fast (ok I know, this one was kind of obvious).

DEMO CLIENT FUNCTIONALITY
The demo clinet is a proof of concept client that is also intended to be useful and usable for anonymization purposes. Therefore the functionality has to be focused on creating solutions.

Design goals:

- Support for SOCKS5 protocol – making it possible to use the client as is with any browser or

software that supports socks.
- Hidden service functionality with distributed DHT for finding the rendezvous points and rendezvous points acting exactly like TCP sockets on the target machine (which is hidden). This would make it possible to use ”hidden” urls on the url bar and have the socks proxy automatically resolve the hidden service and establish a connection to it.

So to break it down a little:
- Support for nested connections from local peer to any other peer.
- Support for outboud connections from the end point of tunnel.
- Support for listening on a local port and creating a circuit for each new connection that arrives there.
- Support for lookup and discovery as well as connection to hidden services published on the network.
- Optional: support for magnet links.

The more people using the software, the more anonymous it will essentially be.

NOTES
The idea of distributed social network can become a reality. It can be like IP but everyone will be able to create new accounts. The moment that you publish something on the network, it gets carried away to all parts of it. ~~Maybe this is a better approach after all – to focus on an anonymous publishing platform instead of a connection network. Maintaining streams and connections does present itself with many practical problems that are inherent to low latency networks.~~ In a distributed publishing store however, you can publish almost anything and all of your messages can be linked to eachother – but maybe not to you directly since you are simply publishing under a public key and not a name. But of course you could publish names and other information together with the data – but that's up to each user who is using the social network.

All the publishing data should be tagged and appropriately categorized right away. Instead of using google you will then use the network itself to find relevant information that you want. This would be like a giant powerful library of hundreds of articles and files published on the network by different people – and you will be able to identify who has published what and search for content published by specific authors by using their public key. And you will also be able to check authenticity of all published content by verifying the signatures. This is the beauty of it because this will allow people to publish freely and yet also retain their credit for the work. Of course you can as has been said earlier, publish stuff under any id – a key can be created on the fly when publishing the content on the network.

Publishing can also be done through an online interface. There will be sites that specialize on publishing content. These sites will be like wordpress blogs – but will have access to the network and so content from them will get distributed to other nodes. A user running a native client and running queries on the database will also cache data for other users on the peer to peer network. Even personal information such as a profile will be published as a ”post” on the network. You basically just ”post” things into the network and then others will distribute the content and deliver it to the people who are looking for it.

Each ”post” has a unique signature that is calculated by signing the post and all comments that may be attached to that post are also ”posts” in themselves and reference this main post. This will allow users to ”lump” together posts. They would for example be able to post a file or an image as a reply to

something else that has been posted on the network and this new ”post” will become distributed through the net exactly the same as the original one, but since it has another post that it is attached to, it may for instance be treated as an attachment or a ”comment”.

These so called ”posts” are in fact packages consisting of HTML, CSS, IMAGES and JAVASCRIPT. A package like that can be created by a piece of software or written manually and then distributed onto the network. So that instead of webhosting, we will have virtually everyone around the globe publishing packages like this. And the most importnat new feature in all this will be that they will all be traceable and have an intristic structure – referencing other packages using unique signatures and ID's. So even if someone does not have a referenced package, they would be able to query the network for that package and then see if someone else has it. If it is found then they will be able to download it and verify it's integrity using the authors public key.

The same system can also be used for publishing software and other digital content. The public key of the AUTHOR does not even need to be found or searched for. The public key is distributed with all the packages along with a package signature. So a user on the network can easily say ”give me more stuff from the same author” and the system will search for all the packages identified by the same key which the author has created earlier and signed all of their packages with.

This brings us to a whole new way of handling dependencies. For example a post can depend on another post and then the system will be able to find that other post and download it from one or more other peers. Of course, if the post can not be found and is not stored anywhere else then it is lost for ever. But the same thing basically happens if you lose the file for some reason or erase it from your drive.

FUTHER FEATURES MADE POSSIBLE BY ENCRYPTION

Encryption and signing does not only make it simple to find more content by the same author, but it also allows people to send entirely confidential messages to eachother using completely public medium. You will for example be able to click a button by a ”post” and click ”reply in private” and then publish the post to the globalnet and the post will travel through countless people and meet countless eyes, but ultimately the only one who will be able to view it is the recipient to whom your reply is intended. Thus private conversations will be entirely private while being conducted in public.

VARIOUS TYPES OF POSTS ON THE ANONYMOUS SOCIAL NET

Posts on the anonymous networks are like ”packages” or packs of data. They can contain almost anything from plain text to images and even javascript. Html and CSS is also possible. You can simply create a pack, sign it and throw it out on one of the publishing servers. It will then get distributed and copied onto many others.

FURTHER INFO

This is a draft outlining the concepts of a distributed, encrypted social internet. It is quite possible to create a cool anonymous network where new concepts can be tested without the need to have any hardware support.

If you have any questions, you can email me to: redbluewebsites@gmail.com

// Martin