Bachelor's Degree Thesis

A New Completely Decentralized Communication System Over IP

Nicholas Helke

August 4, 2013

Prof. Stephan Robert
HEIG-VD

Abstract

The Internet Protocol (IP) was designed to be resilient to topology changes. It should therefore be possible to build a resilient communication system on top of IP, in fact consumer communication systems deployed today all rely to some extend on central servers. This makes these communication systems vulnerable should the central servers become unreachable for any reason.

This memoir proposes a strategy for new communication system over IP that is completely decentralized, i.e. that doesn't rely on central servers at all. This strategy uses the BitTorrent network's Distributed Hash Table (DHT) (which is based on Kademlia) to map users to IP addresses in a completely decentralized manner. An application network protocol and a proof of concept software application which implement the proposed strategy are also discussed in this memoir.

Terms of reference

The purpose of the bachelor's degree project of which this memoir is the report is to produce a new completely decentralized communication system over Internet Protocol (IP).

The key words in [3] are to be interpreted in the context of this chapter as they are defined in said document.

The development of this new system is divided into two parts:

1. A theoretical strategy for establishing a communication over IP in a completely decentralized manner. This strategy must meet the following requirements:
   a) The strategy must enable a conforming implementation to establish a point-to-point connection over IP between two nodes with public Internet Protocol Version 4 (IPv4) addresses where the user need only know the public key of participating node he wishes to contact.
   b) The strategy can not rely on a central server to establish a mapping from public keys to IPv4 address, i.e. the strategy must be "pure" Peer-to-Peer (P2P) according to Definition 2 of [29]
   c) A central seeding server or list of servers may be used to seed fresh installations of a conforming implementation with the necessary information to join the system. Once a fresh node has been seeded however, it must be able to continue to function without ever relying on the seeding server or servers again.
   d) The strategy must be resilient in the face of network splits, i.e. communication must remain possible within the each subnets after the split.
   e) The strategy must enable an established connection to transport arbitrary data (e.g. text, voice, video or files) in both directions between the nodes.
   f) The strategy may employ a variety of other strategies in order to extend the applicability of this strategy to nodes behind firewalls and/or Network Address Translation (NAT).
2. A practical implementation of the above strategy in the form of a proof of concept software application and associated application network protocol. This proof of concept application and network application protocol must meet the following requirements:
   a) The network application protocol must be architecture, platform and language agnostic.

b) The network application protocol must allow for text communication and must be extendible to support arbitrary forms of communication (e.g. voice, video or files). This may be achieved by building feature negotiation into the protocol.
c) The proof of concept application must support Linux and may use any Graphical User Interface (GUI) available on the platforms it supports.

Contents

1. Introduction

It was the Arab Spring that first alerted me to the need for a completely decentralized communication system[1]. The falling dictatorships desperately attempted to thwart the revolutions by cutting and disrupting internet communication[8]. The internet was designed to be and is resistant to topology changes, however the countries of the Arab Spring had a limited number of backbone connections to the outside world, which were all under government control. The failing governments cut themselves off from the rest of the world by destroying the routing tables of the edge routers. This was surprisingly effective at disrupting communications.

The number of people relying on American and European based services had been underestimated, or at least there was no widely known data about the penetration of such services. It turns out Facebook, Gmail, Skype and Twitter where the principal tools of communication for the revolutionaries and so when they were cut off from the rest of the world they were no longer able to communicate internally because they were relying on server-based services based outside their isolated subnet. Despite still being physically connected to one another, they where unable to communicate as they no longer had access to the central server.

A worrying trend amongst all countries developing and developed alike are increasing violations of civil liberties such as [12]. A decentralised communication therefore would also provides some protection against unlawful surveillance although this is more debatable given the governments increased regulation and control of Internet Service Providers (ISPs).

This brought to light the necessity for a completely decentralized communication system over Internet Protocol (IP), one that would be as resilient as IP is itself. It should also be mentioned that such a system would also be beneficial to users on accidental topology changes, such as the recent failure of SEA-ME-WE 4 due to criminal sabotage[1, 2] or cases of accidental failure.

Some effort has gone into structuring this memoir for linear reading. It has not however been possible to eliminate forward-references completely. In order to fully grasp some of the short comings of the solutions in chapter 2 an understanding of some notions of the theory from chapter 3 can be helpful. This forward-reference can be avoided by skipping chapter 2 entirely. It is of no consequence to the comprehension of the solution proposed by this memoir, it serves principally to show an alternative decentralized communication system.

Other P2P solutions of all sorts are explored in chapter 2. In particular, Bitmessage[31], a project with similar intend, is described, analysed and its shortcomings, which prevent

[1]Completely decentralized is to be understood as a "pure" Peer-to-Peer (P2P) system as in Definition 2 of [29].

it from meeting the terms of reference of this memoir (cf. page iii), are exposed.

Chapter 3 covers the challenges of such a communication system. Lessons from past attempts are drawn and the various solutions envisaged in this memoir are described and evaluated.

Chapter 4 describes the process through which a proof of concept application, which is part of the author's bachelors degree, was developed and the protocol which was developed along side it and which others are welcome to use.

2. Existing works

Before embarking on this project I scoured the internet for similar and not-so-much-similar-but-related projects. After sifting through the countless communication systems that call themselves Peer-to-Peer (P2P), but that still rely on some central infrastructure ("hybrid" P2P), I found the following couple of interesting related projects. Although there are most certainly some that I am still unaware of.

2.1. Bitmessage

Bitmessage[31] is a proposal for an anonymous decentralized communication system, first described in a white paper by Jonathan Warren in October 2012. Its abstract reads:

> We propose a system that allows users to securely send and receive messages, and subscribe to broadcast messages, using a trustless decentralized peer-to-peer protocol. Users need not exchange any data beyond a relatively short (around 36 character) address to ensure security and they need not have any concept of public or private keys to use the system. It is also designed to mask non-content data, like the sender and receiver of messages, from those not involved in the communication.

With the exception of the last sentence, the above abstract would aptly describe the proof of concept application described in chapter 4. The two solutions are, however, very different.

Bitmessage is limited by design to non-real-time communication, so is more akin to e-mail—and perhaps a slight regression over push e-mail. This alone makes it incompatible with the terms of reference of this project.

Bitmessage1 is very heavily inspired by Bitcoin[23]. Messages are sent to all the users of the system on a best effort basis, this is necessary for the solution to provide anonymity to both the sender and receiver of messages. In order to protect the system from flooding, which would be catastrophic as all messages must go to all users, proof-of-work must be appended to messages.

Sending messages to all users also serves as the strategy whereby peers find one another. Essentially peers do not find one another, instead they are relatively confident their message will reach its destination as all messages go to everyone. This is clearly not an efficient strategy and one which is more akin to the Gnutella generation (cf. section 3.1 for an explanation of Gnutella's part in P2P systems) of linear complexity flooding protocols of P2P systems than the more recent logarithmic P2P protocols.

This strategy of flooding everyone does however provide plausible deniability. As all messages are encrypted using public key cryptography it cannot be surmised from the message contents alone who the indented recipient is. Given that each message is sent to everyone, determining the recipient by tracking the Internet Protocol (IP) address the message goes to is not possible.

The proof-of-work, which protects the network from flooding, is to take about four minutes on standard hardware. It this proof-of-work also is an effective way of combating spam. Each message requires this four-minute proof-of-work, for the average user this has only a marginal cost in the form of energy costs and slight delay of four minutes in message delivery. Spammers would have to spend those four minutes on each message as the proof-of-work is applied to messages after they have been encrypted with the public key of their respective recipients.

Bitmessage is not a competitor to this paper. It presents an interesting alternative approach, which trades real-time communication for anonymity, plausible deniability and effective elimination of spam.

2.2. Completely decentralised DNS

The major challenge to completely decentralized communication is the mapping of users via some identifier to IP addresses. The Domain Name System (DNS) is certainly the most widely deployed and used system for mapping identifiers, in this case fully qualified domain names, to IP addresses. DNS is also a distributed system, it is not however completely decentralized. In face it is organized in a tree, managed by Internet Corporation for Assigned Names and Numbers (ICANN). The DNS system is therefore subject to the whims of ICANN. Fortunately ICANN takes its responsibility seriously and has not yet censored a top-level domain.

Network Information Centres (NICs) on the other hand have been known to seize domains. In 2010 the United States Department of Justice and Homeland Security started to seize certain dot-com domain names without warning and without possible the possibility of appeal. This has prompted several attempts at creating truly decentralized domain name systems.

2.2.1. Dot-P2P

It appears that the first serious attempt in this field, insofar as it was picked up by the Electronic Frontier Foundation (EFF)[1] was Dot-P2P.

Unfortunately aside from the EFF's mention of it, there is not much I can say about it, as in July 2013, when I was attempting to gather information about it, its website[2] was down, seemingly abandoned.

[1]https://www.eff.org/deeplinks/2010/12/constructive-direct-action-against-censorship
[2]http://dot-p2p.org/

2.2.2. Namecoin

A more recent attempt, and one which has the merit of still being active is Namecoin[3]. Like Bitmessage (cf. section 2.1), Namecoin seems to be following a trend of reusing concepts from Bitcoin[23]. In fact the only way to register a Namecoin domain is to exchange Bitcoins for Namecoins which serve as the currency within the Namecoin ecosphere. Unfortunately I fear its close tie to Bitcoin will ultimately prevent it from ever going mainstream given how controversial Bitcoin is outside its niche. Although, who knows, that may change.

[3]`http://dot-bit.org/Main_Page`

3. Challenges of Completely Decentralised Communication

The use of central servers is inherently vulnerable as a central server's role is to be a known location where information can be fetched or stored. That makes it also a known location which can be blocked, attacked or otherwise disabled. In completely decentralised communication Alice must be able to contact Bob without resorting to the use of a central directory server or list of servers.

In section 3.1 the challenge of finding other peers without using a central directory server is explored.

It is not sufficient to find the peer you wish to talk to, one must also be able to establish a connection with that peer. Unfortunately most internet devices are behind firewalls and/or Network Address Translation (NAT) which are designed to enable devices to reach servers but often completely block connection attempts initiated from devices on the WAN. Section 3.2 discusses the challenges and possible work-around solutions for to the problem of actually establishing a connection to Bob.

Security concerns and possible approaches to address them are the subject of section 3.3.

Any solution to the prior challenges, is not worth retaining as a solution to completely decentralised communication if it is as vulnerable to topology changes as server-based communication solutions. The whole point of completely decentralised communication is to approach the resilience of plain old Internet Protocol (IP) as much as possible. Section 3.4 discusses the resilience of the discussed solutions.

3.1. Finding Peers in a Completely Decentralised Manner

Alice wishes to know how to reach Bob. She cannot use a phone book. She is only allowed to contact random people who may or may not know Bob but who can help her by suggesting other people more likely to know Bob. This in essence is the challenge to finding peers in a completely decentralised manner or a so-called "pure" Peer-to-Peer (P2P) system.

3.1.1. Decentralised data storage

Distributing data over many hosts, a practice known as sharding in the database world, is quite common for large databases. The practice has also been around for some time and is quite well understood. It is however very different and becomes a much harder problem when one does not control the hosts that are to collaborate.

The problem of finding information in a decentralized manner over a collaborating but not centrally controlled set of hosts has been the result of quite some interest ever since the original Napster's collapse. Napster served to illustrate the great power and scalability of P2P systems. It relied however on central servers for indexing the content available within the network. Rüdiger Schollmeier defines such systems as "hybrid" P2P networks in [29]. Napster's collapse revealed a weakness in "hybrid" P2P networks, which rely on central servers to some degree, and led to a series of breakthroughs in P2P data storage.

The first reaction wasn't a breakthrough by any means and it is mentioned here only for historic accuracy. Following Napster's demise the first networks to be born from the ashes did away with central servers by using *request flooding*. Request flooding can hardly be called a strategy, it consists of contacting every peer you have ever known, on the off chance that they are still online and that they have the information you are looking for. Gnutella was perhaps the most popular such system and arguably its only contribution to the field of P2P data storage and retrieval was a study by Saroiu et al of probable future uptime as a function of past observed uptime [28]. The results of this study informed the replication parameters used later in Kademlia[21], the solution ultimately settled on for this project.

A far more interesting development and one that can be called a breakthrough was the development of a range of new Distributed Hash Tables (DHTs). CAN[25], Chord[30], Pastry[27] and Tapestry[34], to name just four. Kademlia also belongs to this set of DHTs, however chronologically it came a couple of years after the aforementioned DHTs and included new features which presented some significant advantages for use cases such as BitTorrent or Emule's DHTs. In particular:

1. It requires less network traffic than the previously mentioned DHTs in order to converge on the requested data.

2. It also dynamically adjusts to load, caching hotspots upstream on the various convergence paths, thereby distributing the load and protecting the random authoritative host(s) from suffering a Distributed Denial of Service (DDoS).

All these solutions have in common the following basic strategy which is still in force today, when developing DHTs:

1. Participating nodes are uniformly spread over some address space. This address space is usually the 160-bit address space and forms what is known as the *overlay network*.

2. Information is spread over the same address space as the nodes usually by using a hashing function and in particular SHA1, hence the 160-but address space.

3. A function is defined that maps data in the address space to the nearest node or k nodes in the address space, where k is known as the replication factor. This is commonly known as the *distance metric*.

4. An algorithm is supplied that given the previous points converges as quickly as possible on the requested data.

The perennity of data in the DHT is dependent on the storage strategy, in particular how it replicates data, and on a sufficient number of nodes remaining online at all times in order for the replication to have a chance succeed. Additionally in a "pure" P2P DHT the number of online hosts must be large enough to ensure that it is highly probable that one of the hosts last known is still online and connected to the rest of the overlay network, as this is the only way that we can rejoin the network while staying true to "pure" P2P. We speak of a critical mass. This is of paramount importance for a system relying solely on a "pure" P2P DHT to be itself perennial.

3.1.2. BitTorrent's Mainline DHT

By far the most widely deployed and used DHT is BitTorrent's so-called Mainline DHT. Because of this, it is the most perennial DHT currently in use. It is the Mainline DHT's popularity and therefore perennity which led the communication system described in this memoir to choose this as a backing DHT thereby ensuring immediate critical-mass. Section 4.3 explains how piggy-backing on the actual Mainline DHT designed for and used by BitTorrent clients for tracking other BitTorrent peers is possible and achieved.

It should be noted that choosing based on popularity for the sake of perennity does not necessarily come at the cost of technical inferiority. It is likely that technical criteria dictated the popularity of the given DHT in the first place. It may come at the cost of the latest and greatest developments in the field, as a popular solution has taken time to become popular and once popular cannot as easily be changed.

In fact, in the specific case of the BitTorrent network's Mainline DHT which chose Kademlia, there are few subsequent DHTs that are technically superior. The principal drawback of Kademlia is its vulnerability to Sybil-attacks, a vulnerability that subsequent DHT designs have attempted to address.

Sybil-attacks are attacks where a large number of malicious peers join a network in order to have a sufficient mass to cause damage or otherwise disrupt service. It used not to be economically viable to perform Sybil attacks, however illegal botnets and legal virtual private servers for hire by the hour make this a far more feasible endeavour. In fact it is rumoured that the big movie studios and record labels hire companies to infest the Mainline DHT with peers that return bad results and/or that log users trying to obtain certain Torrents, data which they later use to report users to the authorities.

A little anecdote about subsequent DHTs and why the final choice for this project remained Kademlia. Petar Maymounkov, one of Kademlia's authors, has since developed Tonika[1] based on the theory in [16], however it is no longer maintained and is not known to be used in any widely distributed solution, so was discarded (despite the reference implementation being in Go, the language chosen in section 4.1).

Another prominent subsequent Sybil-proof DHT is Whānau, developed by Chris Lesniewski-Laas and M. Frans Kaashoek in [18]. It boasts strong protection to Sybil-attacks, up to

[1]`https://pdos.csail.mit.edu/~petar/5ttt.org/`

$O(\frac{n}{\log n})$ *attack edges*.[2] Despite having been tested Whānau on PlanetLab[3], which a chat application no-less, it remains a theoretical exercise for the time being as no widely distributed solution has picked up on it. A shame given that Whānau even boasts constant-time $O(1)$ lookups!

Neither of these promising projects therefore has reached the critical mass needed for it to be of interest to this project.

This is a good opportunity to remind ourselves that ultimately we must produce a usable completely distributed messaging system. This is probably the area where this goal has had the strongest influence in essentially determining the chosen technology, i.e. Kademlia. It is more important that from the get go the solution work and that it work with the inevitable large fluctuations of users in its beginnings. There would be little point in choosing a solution which addresses Kademlia's shortcomings if that solution would then not be sustainable with the inevitably few early-adopters, even if theoretically down the line, with more users, it would work.

In spite of its vulnerability to some distributed attacks Kademlia is quite ingenious. It has already been mentioned that Kademlia has some features not supported by the other mentioned DHTs, and it is easy, in light of these features and the unavailability of the mentioned subsequent works, to understand Andrew Loewenstern's choice for the Mainline DHT in [19].

3.1.3. Kademlia

In order to understand how Kademlia works and how it succeeds in being so efficient, we must first go over the strategic points that all DHTs define:

1. Kademlia uses random 160-bit identifiers which ideally should be completely random in order to ensure uniform distribution, in practive they are pseudo-randomly generated based on the current time, in order to obtain reasonable uniform distribution, and on the machine's MAC address, in order to ensure uniqueness.

2. Data in Kademlia is traditionally spread over the overlay network address space using the SHA1 of the data itself. In the specific case of BitTorrent's Mainline DHT, it turns out that all torrents have a so called *infohash* which represents the data in the torrent and is in actual fact the SHA1 of the torrent's data, leading to a trivial association from any given torrent to the overlay network address space.

3. The bit-wise exclusive-or (XOR) of 160-bit addresses is used as the distance metric for Kademlia. It is this breakthrough choice that enables the additional features that Kademlia provides, as we will soon see.

4. Finally, the convergence algorithm Kademlia uses has logarithmic time-complexity $O(\log n)$ and converges along a constant path, a fact which comes from the prop-

[2] Attack edges are those edges in the overlay network graph, which connect bad and/or malicious nodes, with the sub-graph of good nodes.

[3] `https://www.planet-lab.org/`

erties of the XOR distance metric and which enables the caching of data upstream from the authoritative nodes, thereby distributing the load and preventing DDoS

Let us first note that bit-wise XOR is a valid distance metric:

$$d(x, y) = x \oplus y \tag{3.1}$$

$$d(x, x) = 0 \tag{3.2}$$

$$d(x, y) > 0 \Leftrightarrow x \neq y \tag{3.3}$$

$$d(x, y) = d(y, x) \quad \forall x, y \tag{3.4}$$

which also respects the triangle inequality:

$$d(x, y) + d(y, z) \geq d(x, y) \oplus d(y, z) \tag{3.5}$$

$$\wedge\ d(x, y) \oplus d(y, z) = d(x, z) \tag{3.6}$$

$$\Rightarrow d(x, y) + d(y, z) \geq d(x, z) \tag{3.7}$$

Of more interest to us however is the following property, which makes Kademlia scale very well. Given x, a node, and δ, a distance, there exists only one node, y, such that:

$$d(x, y) = \delta \tag{3.8}$$

Chord and the seminal Kademlia paper call this property being *unidirectional*. This property combined with Kademlia's routing algorithm means that any given node tends to converge on requested nodes along different but constant paths. This means that nodes closer to the querying node along the convergence path can cache the data thereby protecting the node closest to the requested information and therefore the authoritative node from DDoS. In order to better see how this works, it is time to review how Kademlia converges on peers.

Kademlia's overlay network can be seen as a binary tree where each successive level represents sucessisve bits of the nodes' IDs. In keeping with reality not all IDs are used and therefore we represent the tree stopping at unique prefixes. This also aids us in fitting the tree on the page.

This tree is depicted in Figure 3.1. We note that in this binary tree the position of highest order bit of the xor-distance which represents the length of the common prefix between two nodes corresponds with how far up the tree one must go to reach the other node.

Next we introduce an extremely important concept for Kademlia's node-finding algorithm. Each and every node in the Kademlia overlay network views and sub-divides the complete tree differently. For the purposes of this example, we will consider a node with unique prefix $0011\cdots$. In figure 3.1 this node is marked in black and the sub-trees $0011\cdots$ divides the tree into are surrounded by grey ovals. We note that these sub-trees correspond to nodes with prefixes differing in successive bits. Note also in descending the tree along the chosen node's path, how each sub-tree is an order of magnitude smaller than the previous one.

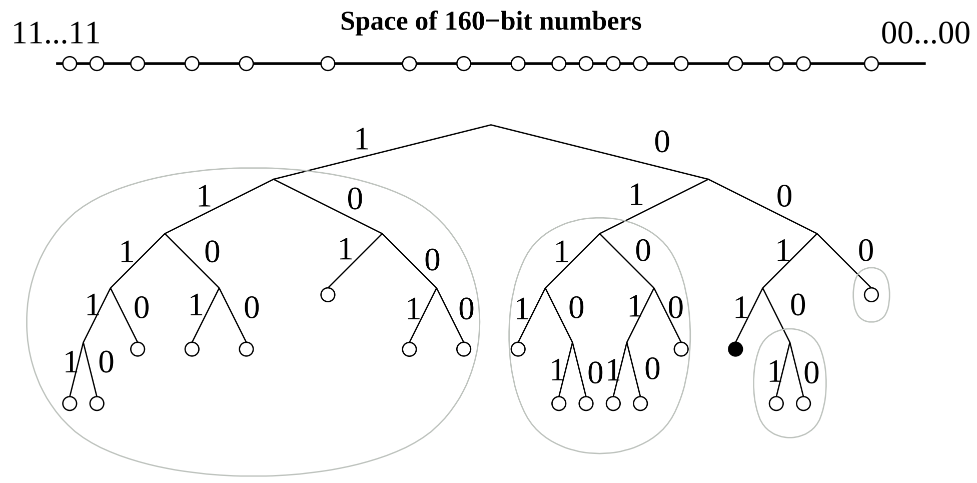


Figure 3.1.: Kademlia binary tree. The black dot shows the location of node $0011\cdots$ in the tree. Grey ovals show subtrees in which node $0011\cdots$ must have a contact. (Source: [21])

We now note how if each node only knows a single node for each of its individual sub-trees that is sufficient to be able to find any other given node.[4] As every node has divided his immediate neighbourhood into smaller sub-trees, if the given node we know in a given sub-tree is not the node we are looking for, we are certain that the node we do know will know a closer node to the node we are looking for (if it doesn't know it directly) as it has divided the tree according to its location.

In actual fact, in order to work around so-called node *churn*, each node tracks k nodes in each sub-tree. These lists are known as *k-buckets*. k is known as the replication factor and is chosen to ensure with high probability that at least one of the nodes will remain online at any given time. Additionally the list is curated periodically to eliminate unreachable nodes and resort the list by uptime of each node, as [28] has shown that the longer a node stays online, the longer it is likely to remain online.

An example of the path taken to converge on a given address is illustrated in figure 3.2. We note how our starting node, $0011\cdots$, is looking for a node, $1110\cdots$, but only knows $101\cdots$ in the k-bucket where the highest order bit differs. $0011\cdots$ contacts $101\cdots$ and asks him if he is the closest active node to $1110\cdots$ or who he knows who is closer than him to that node-ID. $101\cdots$ only knows $1101\cdots$ in his corresponding k-bucket and he returns this to $0011\cdots$. In turn $1101\cdots$ returns the only node known to him, $11110\cdots$. Finally upon contacting $11110\cdots$, $0011\cdots$ gets the address of node $1110\cdots$.

The only networking requirement of Kademlia is P2P User Datagram Protocol (UDP) connections.

[4]We will not reproduce the proof here, it can be found in [21].

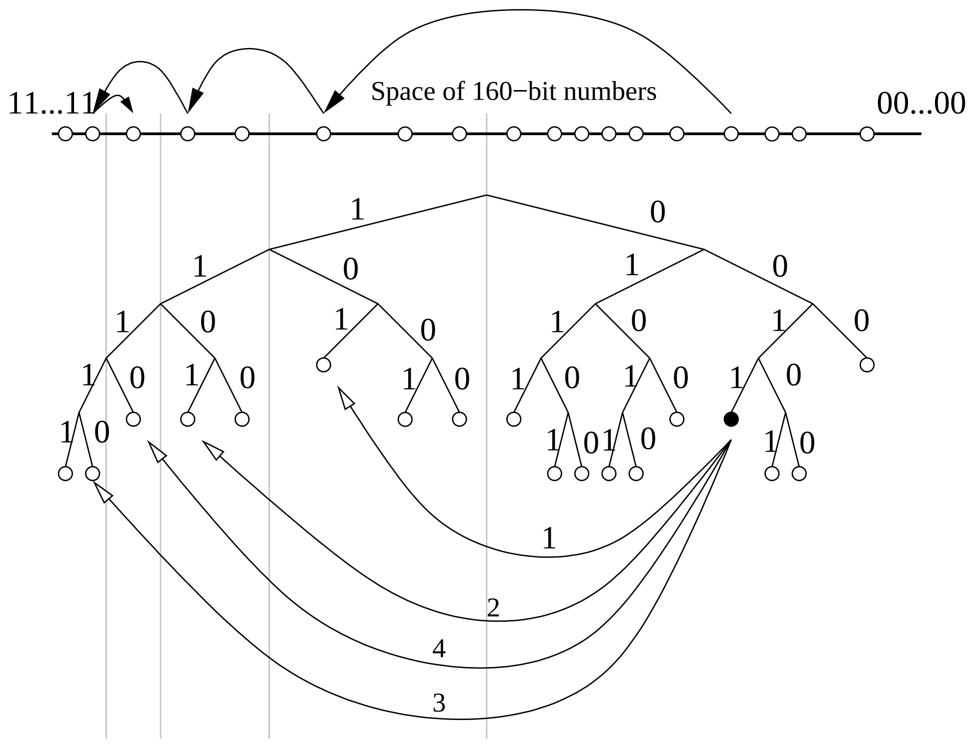


Figure 3.2.: Locating a node by its ID. Here the node with prefix 0011 finds the node with prefix 1110 by successively learning of and querying closer and closer nodes. The line segment on top represents the space of 160-bit IDs, and shows how the lookups converge to the target node. Below we illustrate RPC messages made by 0011. The first RPC is to node 101, already known to 0011. Subsequent RPCs are to nodes returned by the previous RPC. (Source: [21])

3.2. Establishing the connection

Having found Bob in the proverbial haystack that is the internet, Alice is going to want to establish a connection with him. The choice of the term communication is deliberate as once a connection is established (and depending on the bandwidth), Alice and Bob can exchange anyway they want. Obvious choices are text chat, Voice over IP (VoIP) or video. This memoir will only provide text as the expertise and codecs needed for VoIP or video fall outside the scope of this memoir. Regardless of the chosen medium of communication or the IP version used, a majority (to not say all) users are located behind firewalls, designed to refuse uninitiated incoming connections. This makes establishing "pure" P2P connections a real challenge, a challenge which is the subject of this section.

There is or was some hope that this would no longer be an issue when Internet Protocol Version 6 (IPv6) finally gets widely deployed, as each device will in theory be directly addressable from the internet and NAT will no longer be an impedance to establishing connections. Sadly early indications are that firewalls built into IPv6-compatible consumer routers will by default refuse uninitiated connections, meaning that the same contrived strategies developed to circumvent Internet Protocol Version 4 (IPv4) NAT, will unfortunately continue to be relevant after the switch to IPv6, in order to allow incoming connections.

The terms of reference of this memoir call only for IPv4 connectivity, so we will focus on that in this section, we should however keep in mind that some of this work will be relevant should this project ever be extended for IPv6 connectivity.

The cleanest means to enable P2P connections to be established through NAT and arguably the only means that respects the definition of a "pure" P2P system, is the use of port mapping protocols to negotiate the opening of incoming ports on the router or Internet Gateway Device (IGD) and to discover the publicly addressable IP address that should be announced to the world. There are two main port mapping protocols, Universal Plug and Play (UPnP) and Port Control Protocol (PCP), both are discussed in more detail in section 4.4.

Not all IGDs support port mapping protocols. These protocols are considered by many to represent security risks and so are generally disabled (even if the hardware supports them) on most (to not say all) enterprise IGDs. As this limitation concerns all P2P networks, both "pure" and "hybrid", wishing to establish P2P connections between peers, other solutions have already been developed, however there is no panacea.

Session Traversal Utilities for NAT (STUN), defined in [26], is a protocol enabling nodes behind NAT to discover not only their publicly addressable IP address but also their external port number. This information is only valid however for non-symmetric NATs as defined in [20]. Additionally STUN's reliance on servers discovered using Domain Name System (DNS) service records, means that using this strategy immediately downgrades any P2P network to a "hybrid" P2P network.

Some of the principals of STUN can potentially be used in a "pure" P2P system as long as each peer acts as a STUN server. There are however security concerns with such a strategy. An unauthenticated STUN server could potentially perform a DDoS attack by providing all nodes with the IP address of the victim as their own, the nodes would in turn publish this to the DHT thereby getting other nodes to DDoS the targeted IP address. If we only perform STUN procedure after we have authenticated the peer as trustworthy (a concept we will discuss in section 3.3), we can then only realistically join the network as long as one of our contacts remains not only online, but accessible in case of network split, meaning he must be in the same subnet after the split. These make for unrealistic expectations and reliance on such a strategy would weaken the system's resilience as a whole.

It should also be noted as long as one of the two peers which are attempting to establish a connection is publicly accessible (perhaps thanks to port mapping) a connection can be established between the two. It is only if both users are not publicly accessible that additional methods must be employed.

Commercial communication solutions use Traversal Using Relays around NAT (TURN) as defined in [20], or in plain English, a *relay server*. As with STUN while the standard is designed around the use of a server, the principals can be reused in a decentralised solution, although it may no longer strictly be considered a "pure" P2P solution, according to [29], as the peers are no longer all equal. This is by no means a novel idea; Skype uses them. In fact relay servers in a decentralised system are known as *super-peers*.

As noted above as long as one of the nodes is publicly accessible, a communication can be established. The addition of super-nodes therefore enables two non-publicly accessible nodes to nevertheless communicate by using a third-party super-node which is publicly accessible as an intermediary for the connection.

Super-peers are far from ideal. Skype was infamous for automatically using clients running their software as super-peers or *supernodes*, without warning their users or asking for their consent beyond an article in their terms of service. A more conscientious way, would be to ask potential super-peers whether they are willing to act as such, thereby allowing them to reach this decision consciously and to factor in considerations like their bandwidth costs. This of course will mean that only a small minority of the minority of potential super-peers will actually play that role, as many users unfamiliar with the concept will, in doubt, simply turn down the option, if given the chance. This probably factored into Skype's decision to simply require potential super-peers to act as such in their terms of service.

The observant reader will have noticed the use of the past tense in the last paragraph. The reason is that Skype has since abandoned its super-peers, as reported in [11], in favour of centrally controlled relay servers that act like the old super-peers.

Another downside to super-peers is that they potentially open up the protocol to eaves-dropping. An obvious solution is to use encrypted connections, thereby theoretically preventing super-peers from knowing the contents of what you exchange with other peers. However in an age where governments are building super-computers with the potential to brute-force encrypted connections in a matter of minutes, governments might be tempted to set themselves up as super-peers and thereby eaves-drop on many communications within the network. Additionally in case of political tension, the governments could easily turn off their super-peer network thereby disrupting communications on the network, if the network grew to rely on super-peers. Latency is another issue with the use of relays, particularly for VoIP or video. The additional encapsulation required to route the packets via a relay would add to the latency induced by the lower network layers.

Interestingly ever since Skype has centralized its super-peers, it has started sharing data with the National Security Agency (NSA) as part of PRISM, as reported in [22].

Theses considerations mean that using super-peers actually has the potential to weaken the resilience of the network. While it may enable more users to connect to the network in the short-term it also increases greatly the number of users who would suffer service disruptions if the network were to be disrupted. Section 3.4 goes into more detail of the resilience of the chosen strategy to topology changes.

For the sake of exhaustiveness, I must point out that in light of recent revelations about the practices of western governments' security services, it would seems that even western governments have control over the Internet Service Providers (ISPs) that serve most if not all end-users. This means that resilience to topology changes might be of little relevance, if the government can simply tell ISPs to shut down. Concerns over privacy and resilience with super-peers pale in comparison to theses concerns. However the solutions to these concerns of government over-surveillance are an issue for a political science paper and not a computer science issue.

3.3. Security

We have already discussed in section 3.2 the challenges and potential issues with passing your data around third parties and how encryption could be used to help protect against such attacks. Encryption is also just generally a good idea. There is no reason to offer would-be eavesdroppers the opportunity to do so. In this section we will discuss possible encryption techniques and the possibility of obfuscating the protocol.

This is not however the only security concern. As with any P2P communication system that maps users to IP addresses caution must be taken to prevent third parties from tracking the location of users by tracking the IP addresses they connect from. In this section we will therefore also discuss what steps we propose to mask to a certain extend this information.

3.3.1. Obfuscation

The use of Mainline DHT, as discussed in section 3.1, means that we announce ourselves to BitTorrent clients as a BitTorrent client participating in the swarm with the infohash (the BitTorrent terminology for a Torrent's SHA1 hash) of our public key. This means that a non-standard BitTorrent client which has been programmed to attempt to download any Torrent it hears about through the DHT might attempt to connect to our client. This fact actually gave me the idea that I might down the line add an actual BitTorrent client to the program and offer up the actual public key file if a BitTorrent request was received, easily identifiable by the protocol header, cf. figure 3.3.

This simple and plain text header coupled with the enormous amount of traffic gen-

```
00000000  19 42 69 74 54 6f 72 72  65 6e 74 20 70 72 6f 74  |.BitTorrent prot|
00000010  6f 63 6f 6c                                       |ocol|
00000014
```

Figure 3.3.: Hex dump of BitTorrent protocol header as defined in [6]

erated by BitTorrent prompted ISPs to throttle BitTorrent traffic[5].

In response some BitTorrent clients (but notably not the reference implementation) introduced Message Stream Encryption (MSE)[6]. MSE uses Diffie-Hellman key exchange, the infohash of the torrent the connection's initiator is attempting to download as a shared secret and random noise to obfuscate traffic between BitTorrent clients including importantly the headers. The solution uses the exchanged key to encrypt the rest of traffic using RC4. The solution is not cryptographically secure. It is designed to be a cheap (resource-wise) obfuscation solution and nothing more.

I toyed with the idea of using the same method so as to provide users of the system with plausible deniability, if for some reason the use of this system was to become illegal someday, somewhere. Statistical detection, it transpires in [15], is capable of detecting BitTorrent traffic even with MSE.

Consider voice communication. Talking is made up of moments of sound separated by pauses, particularly in conversation when the speaker is considering what to say on the spot. An additional tell-tale sign of voice communication is that generally only one person speaks at a time. Tracking the volume of the packets send between hosts enables one to recognise these patterns of silence, talk and only one user at a time. In fact Charles V. Wright et al show in [33] that it is not only possible to recognise the use of VoIP even over encrypted channels, but also to detect certain spoken phrases with characteristic patterns.

This statistic detection means it would very likely also be possible to differentiate the traffic of the proposed communication system from that of BitTorrent traffic rendering null and void the possibility of plausible deniability. Unfortunately I came by this information only after I had already written and submitted a patch (cf. appendix D.2) to a Go library for Diffie-Hellman key exchange[7] which was missing support for arbitrary parameters not defined in the seminal Requests for Comments (RFCs) [14, 17].

As mentioned above, MSE is not cryptographically secure, so its use would have added an additional layer of encryption to the layer necessary to ensure private communication. Given the statistical detection and the fact that the traffic would in any case have to be encrypted by other means as well, I decided to abandon the idea of obfuscating the traffic. Steganography is therefore left as an exercise for the reader.

[5]`https://torrentfreak.com/new-data-exposes-bittorrent-throttling-isps-120809/`
[6]`http://wiki.vuze.com/w/Message_Stream_Encryption`
[7]`https://github.com/monnand/dhkx`

3.3.2. Handshake

Anyone can announce themselves as you on the Mainline DHT, by design. One of the first things I tried when I had the DHT code working was looking up the hash of my public key, which I am fairly confident does not correspond with a real Torrent. I discovered that countless peers announced themselves as participating in such a torrent, even though the chance of such a Torrent existing is less than your chances of being mauled by an escaped gorilla[8].

It turns out that there are numerous bad nodes out there, some sponsored by record companies, others by movie studios, and who knows who else. The roles of these bad nodes can be to pollute the DHT with incorrect information, track usage, a combination of the former and the latter, or something else yet.

This means that when we search for a given person, we cannot be certain which of the many hosts that announce themselves are genuine. Obviously the only way to be sure is to attempt a handshake with each and every peer. Some will simply drop the connection, other will mimic the BitTorrent protocol, and there may well be other, as yet, unobserved behaviours. Regardless we must design the handshake in advance to protect against malicious peers that might be set up to track users. The handshake should also be hardened against *replay attacks* which would-be spammers might be tempted to exploit in order to send spam on the network, should it ever garner a sufficient following.

A good handshake should therefore ensure the following:

1. The handshake must protect the initiator from identifying himself to an unauthorized user who might otherwise impersonate one of your friends in order to track you by logging your IP address every time your node attempts to connect.

2. The handshake receiver must be reasonably protected against replay attacks, where an unauthorised host who has somehow capture an initiator's packet sends it to the receiver in the hopes of discovering if the receiver is present and whether the receiver responds to the captured initiation packet.

3. The handshake initiator must be reasonably protected against replayed responses, where a malicious host attempts to fool the initiator into believing it is an authorised peer by replying with a previously captured good response from a real authorised peer.

Meeting these requirements can be done using public-key cryptography, by following the following procedure:

1. The handshake initiator signs his/her public IP address, the current date and time in Coordinated Universal Time (UTC) and a random authentication token with his/her private key and then encrypts message with the public key of the receiver.

2. The receiver decrypts the message, checks the signature. To guard against replay attacks, the receiver checks the IP address and the timestamp.

[8]http://stackoverflow.com/questions/4676828#4681221

3. The receiver replies to the initiator with his IP address, timestamp and the random token received from the initiator, signed with the receiver's private key and then encrypted with the initiator's public key. At this point the receiver is reasonably certain of the receiver's authenticity and starts awaiting messages from the initiator.

4. The initiator checks the IP address and timestamp of the returned packet as well as the random token sent to the receiver. If the tests pass, the initiator can be reasonably certain of the authenticity of the receiver and can start exchanges messages with him/her.

The use of the tuple (IP address, timestamp, random token) means that we can safely relax the timestamp resolution to one hour thereby removing the need for synchronized time between hosts.[9] This tuple forms the main protection against replay attacks, it is unlikely in any given hour that an attacker will be able to take over a given IP address and see the same random token be generated.

As all messages in the handshake are encrypted with the public keys of their intended recipients, the best an attacker could hope to achieve with a replay attack is user tracking. That is an attacker could confirm the location of a given user by replaying the packets of a contact of that user. The proposed system cannot be expected to detect cases where the private key has been compromised.

3.3.3. Encryption

Public-key cryptography is computationally expensive. Like many before, we propose using public-key cryptography only for the handshake. Thereafter Advanced Encryption Standard (AES) should be used. The attentive reader will recall our handshake procedure includes a random token. This random token corresponds to the AES key used for all communication after the handshake.

You will also recall that at the end of our handshake procedure the receiver is in the waiting state, waiting for messages from the initiator. The initiator must therefore send a ping over the AES channel if the handshake completes to tell the receiver that the AES channel is established and symmetric, meaning that either host may send messages on the channel asynchronously. The actual network protocol is discussed in section 4.6.

3.3.4. Web of Trust

In order to prevent spam, Web of Trust (WoT) is used in lieu of expensive and environmentally unfriendly proof of work as used by Bitmessage (cf. section 2.1). This does however mean that new people may not contact you before exchanging their public keys with you. This strikes me as an acceptable trade off. Additionally I have no plan

[9]It dawns on me that time synchronisation between coordinating but not centrally controlled hosts would make for an interesting paper. The problem of time synchronisation between centrally controlled hosts is a well known textbook example.

to replace the existing Pretty Good Privacy (PGP) key-servers. In case of degraded connection, this may prevent temporarily the creation of new contacts. A possible work around for this would be manually exchange keys, using physical media like a memory card or stick. Another option, and one which could conceivably be build into the application at a later date, would be to use the BitTorrent protocol, seeing as nodes announce themselves as participating in the torrent corresponding with the public key. We have already seen in section 3.3, that BitTorrent has a distinctive header. It would be relatively easy add a minimal BitTorrent client to the program that would serve up the public key, upon detecting a BitTorrent connection. This option however would be reserved as a stop-gap solution for those cases when the standard PGP key-servers are unavailable.

3.4. Ensuring resilience to topology changes

Alice is connected to many different peers participating in the communication system. All of a sudden a backbone connection is lost. Alice now wishes to contact Bob. A physical link exists between Alice and Bob, however who is to say whether Alice knows random people she can contact between her and Bob who are in the same subnet formed by the lost uplink connection.

It turns out unfortunately that this is impossible to predict and will depend on many different factors outside of the control of this project. One worrying fact even without topology changes is that [7] found that due to variations in implementations of routing code not all the supposedly compatible Mainline DHT clients conform sufficiently to Kademlia in order to ensure deterministic convergence on requested nodes.

We can attempt to anticipate what might happen, however without a full scale test this will remain purely theoretical behaviour. What should happen when the network splits, is that users on either side of the split will all of a sudden see a number of the nodes they have in their k-buckets disappear. It will and should be able to compensate for this provided that at least one node per k-bucket is still accessible. Since all nodes will behave in this manner, the result will be two completely independent Kademlia DHTs. One on each side of the split.

For instance without tests it is difficult to estimate how probable it is for more than two DHTs to exist after the physical split. Intuitively it seems unlikely, however theoretically it could still happen.

Consider three subsets of nodes A, B, C. Before a topology change, they are all physically connected to one another, however it just so happens that nodes in A only have nodes from $A \cup B$ in their routing tables, while C only has nodes from $C \cup B$ in its routing tables. If we now split the network between A and B and C and B such that A and C are still physically but A is not, we have created a case where there are three overlay-subnets for only two physical subnets.

Additionally these overlay networks may not be able to join back together automatically. The most effective way to merge split overlay networks is to reseed routing tables using a common centralised DHT node. Under normal circumstances this seeding server is used by freshly installed clients to enable them in seed their routing tables. It is however reasonable to assume that if a network split occurs these are not normal circumstances. Theses seeding routers, like all central infrastructure are a weak link, and could relatively easily be attacked in coordination with a network split so as to prevent the networks from merging without a software update changing the seeding server address.

While I was at San Jose State University (SJSU) finishing this thesis, I came across a platform, called PlanetLab[10], for testing P2P programs like this one. I tried to take advantage of it while I was in San Jose. Jon Pearce, dean of computer science at SJSU, kindly put me in touch with the engineer responsible for SJSU's participation in PlanetLab, Xiao Su. Unfortunately she was away while I was in San Jose and was unable to grant me access to the platform while on the road. I was therefore most unfortunately unable to confront the above theory to reality.

In summary the theory is that a network should split into just two overlay-subnets and the two networks should be able to merge again provided enough time and common the seed of nodes. Unfortunately without PlanetLab it was impossible to test.

[10]`https://www.planet-lab.org/`

In theory, theory and practice are the same. In practice, they are not.

(Albert Einstein)

4. Implementation

This chapter discusses the implementation of a proof of concept software application implementing the theoretical solutions discussed in Chapter 3 and an associated network application protocol.

The implementation details are discussed through concrete examples in the Go programming language[1], which was used in the development of the proof of concept application. The application network protocol is however, as required by the terms of reference (cf. page iii), architecture, platform and language agnostic. It should therefore nonetheless be possible to implement a conforming application in another language by following this chapter.

The process by which the application's architecture was determined and the reasons behind the choice of Go as a programming language are discussed in section 4.1.

The shaping of the reference implementation's internals are the subject of section 4.2.

Section 4.3 discusses the use of Kademila[21], a Distributed Hash Table (DHT), to find the Internet Protocol (IP) addresses of node in the network. More specifically the section discusses the use of the BitTorrent network's instance of Kademlia, known as the Mainline DHT [19].

Section 4.4 discusses the connection process, a Go library I wrote for the purpose of mapping ports on Universal Plug and Play (UPnP) Internet Gateway Devices (IGDs) and confronts the theory of traversing Network Address Translation (NAT) with reality.

Section 4.5 discusses the implementation of the security features discussed in section 3.3 and in particular the reworking of the handshake procedure to work around NAT.

Section 4.6 discusses the communication protocol and its features to support arbitrary communication mediums including but not limited to text, voice, video and files.

4.1. Designing the program's architecture and choosing a programming language

The strategy described in Chapter 3 has a significant drawback in that it is slow. Median lookup times were found in [7] to be over a minute. Although in my experiments I found it to be much faster, often less than 30 seconds. This still makes for an unacceptable launch time for an interactive user oriented application in this day and age. This led me to consider decoupling the application into two parts, a daemon which starts with the user session and a front-end application which merely presents the communications and availability information collected by the daemon.

[1]`http://golang.org/ref/spec`

A considerable advantage to decoupling the backend and the frontend is that it forces the creation of a well defined Application Programming Interface (API) between the two. A well defined API greatly simplifies the creation of a multitude of different frontends for a wide variety of device and platforms. This in turn encourages the development of native frontends on each platform, which greatly improves the user experience on each platform, thereby increasing the good-will towards the application. An important consideration for increasing an open-source application's audience, where there is no money for marketing.

Splitting the program into separate daemons and front-ends is also necessary in order to efficiently support smartphones. While far more efficient than BitMessage (cf. section 2.1), the strategy of this paper still is not appropriate for mobile applications where power efficiency is the primary concern. The number of connections required and the fact that the application must keep a socket open for incoming connections makes it a poor match for mobile devices.

Client-server communication is far more efficient than Peer-to-Peer (P2P) communication. Decoupling the back-end would allow a mobile client to be developed which would let the back-end run on a server. Obviously this defeats the whole purpose of this communication system, however it can be seen as just one possible tier in a tiered solution.

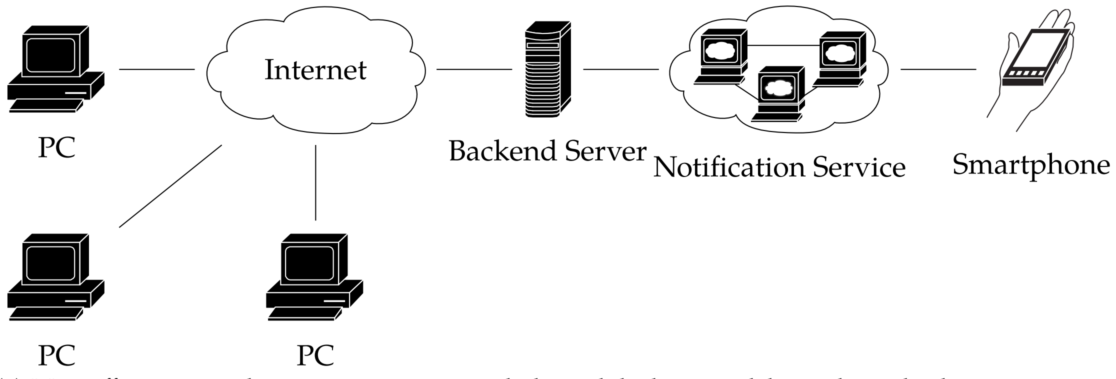
The tiers can be organised from most energy efficient and least resilient to least efficient and most resilient. Users not concerned with delegating their backend process to a third party could opt for an even more efficient tier which could use vendor provided push notifications services to further increase energy efficiency on the mobile device. The three tiers are described below and illustrated in figure 4.1.

1. The bottom tier from a reliability perspective and the top tier from a power efficiency perspective is to delegate the running of the daemon to a central server controlled by the mobile application developer. The central server can then use vendor controlled push services (e.g. Apple Push Notifications[2] or Google Cloud Messaging[3]) to push events to the client as needed, leaving the application free to suspend completely when not in use. Figure 4.1(a)

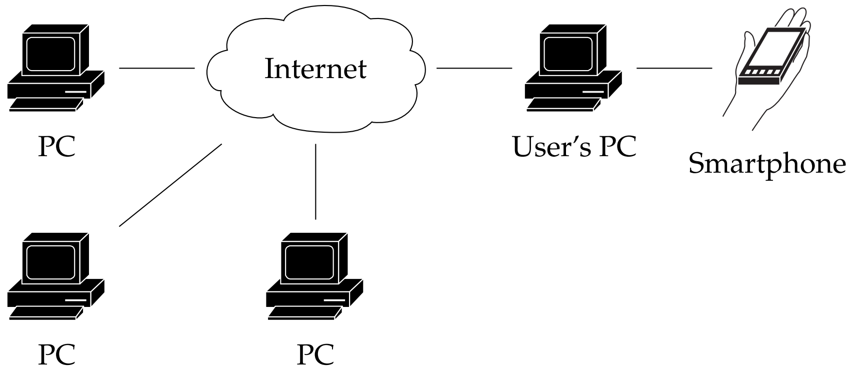
2. The second tier would be for the mobile application to connect to an instance of the daemon running on a machine controlled by the user (e.g. his desktop computer). This would then only require the mobile application to maintain one connection, with the daemon. This would require the application to stay in the background, thereby draining the battery more than in the first tier. Figure 4.1(b)

3. The third and most resilient tier would be used as a fallback should either or both of the first two fail and would consist of running the daemon on the mobile terminal. Figure 4.1(c)

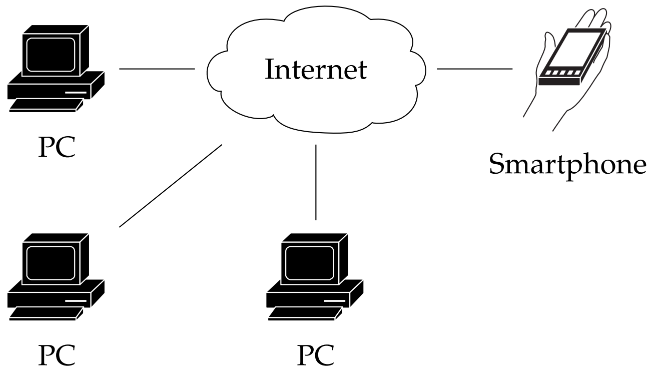
[2]https://developer.apple.com/notifications/
[3]https://developer.android.com/google/gcm/



(a) Most efficient tier, where communication with the mobile device is delegated to a third party



(b) Efficient tier, where the mobile device keeps a single connection open with the user's own PC, which runs the backend



(c) Most resilient tier, where the phone runs the backend locally and connects directly to the other clients of the communication system

Figure 4.1.: The various possible tiers which can be used by a mobile device to participate in the communication system from most efficient and least resilient to lest efficient and most resilient

Ideally a mobile implementation of this communication system would provide all three tiers and would automatically switch between them, or at least let the user switch between them as they become viable or not.

Initially I thought of using ubiquitous C. C has the advantage of being very portable. C even works on Android, using the Native Development Kit (NDK). C also has a vast array of well supported (and sometimes well written) libraries for a wide range of use cases.

C also has considerable disadvantages and is a ruthlessly low-level by today's standards. It is neither type-safe nor memory-safe, almost inevitably leading to segmentation faults during development (or worse during deployment) which are a real bane to debug.

I also considered two Java Virtual Machine (JVM) languages, Java or Scala[4]. Both Java and Scala are memory-safe and Scala is type-safe to boot. I ultimately ruled them out as too bloated for a daemon and because Apple is no longer shipping the Java Runtime with new Macs thereby severely diminishing its attractiveness as a cross platform solution.

I ultimately settled on Go.

Go is a relatively young open-source system programming language developed by Robert Griesemer, Rob Pike and Ken Thompson at Google starting in 2007. It is both memory-safe and type-safe, it has garbage-collection and sports a C-like syntax. It is a fast compiled language which produces statically linked binaries only slightly larger than C binaries that link to the local *libC*, but are still an order of magnitude smaller than what would be possible with the JVM. It is simple (thanks to [4]) to cross compile Go programs for all the platforms it supports and, importantly, it provides a unified network interface regardless of the kernel. The only relevant platform not yet supported by Go is Darwin/ARM, however given that there is no facility to run daemons on stock versions of iOS. It can therefore be argued that this is a non issue for this project as only the first two tiers are possible in any case on that platform.

My inner autodidact also wanted to try this language that it had learn the previous summer but had not yet put to good use.

4.2. Reference implementation's application architecture

The terms of reference call for a reference implementation. We have already discussed in section 4.1 generally our plans for the reference implementation. In this section we will discuss the plan for the application's inner structure and then in the subsequent sections we will actually see the code.

Now is also the time to announce that the codename of the reference implementation is **Dictator Breaker**. You will see this mainly in the code in appendix B.

One of the reasons I chose Go is its notoriety for good, compiled, cross-platform support, as covered in section 4.1. There are however some platform specific issues that no abstraction layer can efface. In particular there are no so-called UNIX sockets on

[4]`http://www.scala-lang.org/`

Windows. This is an issue for communication between the backend and the frontend; a UNIX socket would have been ideal for this.

In order to work around this issue I decided to go a different route and develop a WebSocket server bound to the loopback interface, that would serve the Graphical User Interface (GUI) to users in their browsers. Serving a GUI as Hyper-Text Markup Language (HTML)/JavaScript (JS) over Hyper-Text Transport Protocol (HTTP) is not new or indeed scarce. Probably the most widely deployed such program is the Common Unix Printing System (CUPS), which is installed on all Macs and is the principal printer subsystem in use in Linux-based distributions.

Another platform specific note is for tiered solutions for smartphones. For the purposes of simplicity the reference implementation includes the encryption in the backend to avoid having to implement it for each different platform in each different backend. This works well on desktop machines where the communication between the backend and the frontend happen over the loopback interface and therefore are safe from snooping (except from root). In the tiered solutions proposed for smartphone communication in Figure 4.1 the channel between the backend and the front end is not secure except for 4.1(c) which is equivalent to the desktop case. In order to reduce the need to substantially modify the backend for the smartphone use-case, I structured divided the program into the components seen in figure 4.2.

The plan was that only the Protocol Manager would have had to be adapted for the mobile use-case and that the interfaces, be they desktop or mobile, would sit just above it.

If you look closely you will note a circular dependency in the graph of figure 4.2. This circular dependency and the fact that the interface turned out to need to influence happenings as far down as the connection manager led me to the realisation that the structure didn't properly match what I needed for the data and events that I was processing.

In stead I opted for a monolithic event or run loop. This loop, in the main function would take care of all incoming events, be they from the OS or from the GUI and would dispatch them as necessary.

The structure and packages in the final architecture are illustrated in figure 4.3.

4.3. Finding peers

Something I found amusing when it came to implementing the finding of peers using BitTorrent's Mainline DHT is that, for the first time I can remember, I was unable to find a maintained C library. This fact played a role in the choice of Go. The best I could find for C was BitDHT by Robert "DrBob" Fernie, however it was last updated in 2002. I contacted DrBob who informed me that he had moved on to other projects.

Finding a well maintained self-contained library to interface with the Mainline DHT of BitTorrent for Go, turned out to be a leisurely walk in the park. I searched for "dht" on GoDoc[5] and immediately came across a "dht" package maintained by Yves Junqueira

[5]`http://godoc.org/` – GoDoc is a central website which keeps up-to-date documentation on all open-

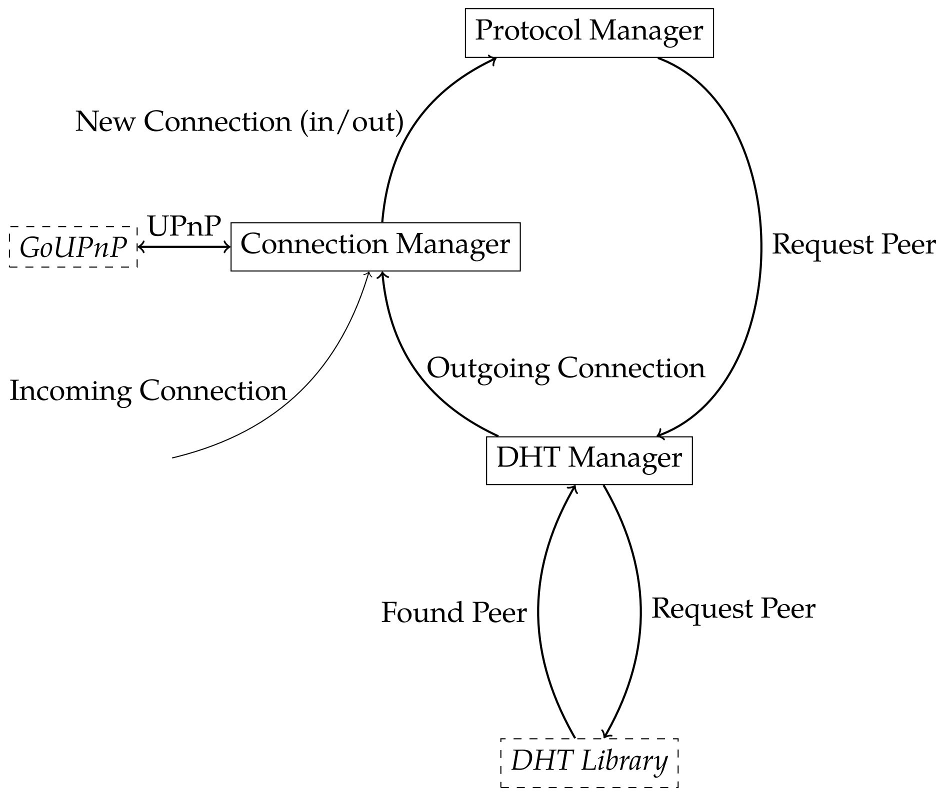


Figure 4.2.: The various components of the original plan for the reference implementation and the messages that they exchange. $\boxed{\text{Solid rectangles}}$ represent the various components of the originally planned reference implementation. Arrows represent OO messages passed between components. Arrows not originating in a component, are external events, passed to the program by the OS. *Italic dashed rectangles* represent external libraries.

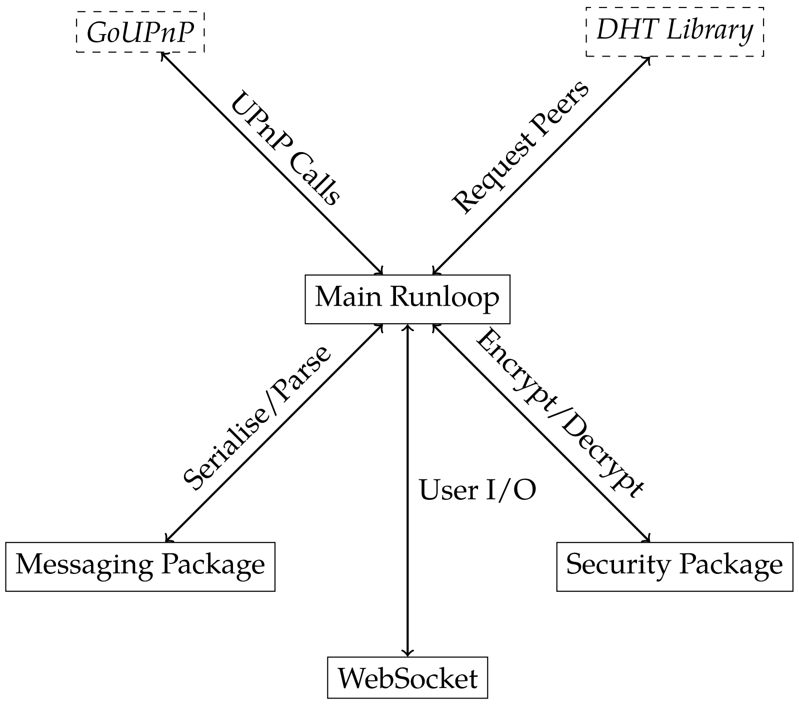


Figure 4.3.: The ultimately settled upon architecture. $\boxed{\text{Solid rectangles}}$ represent the internal packages of the application. *Italic dashed rectangles* represent external libraries. Arrows represent what the $\boxed{\text{Main Runloop}}$ exchanges with the other package or it uses them for. All OS events are handled by the $\boxed{\text{Main Runloop}}$

on GitHub[6]. At the time there was only one other result which was completely inactive and incompatible with BitTorrent's Mainline DHT.

The package's interface is remarkable simple. A minimal example to get up and running might Go (pun intended) something like this:

```
// Create a new DHT instance
// Arguments are:
//    - local port number
//    - target number of peers in routing table
//    - a boolean indicating whether to store the routing table on disk
d, err := NewDHTNode(6881, 100, false)
if err != nil {
    fmt.Println(err)
    return
}

// After checking above that there was no error, we can safely fire up the
// DHT's thread
go d.DoDHT()

// Now we can search for a particular key or infohash in BitTorrent parlance
// Arguments are:
//    - the binary infohash stored as a '\0'-terminated string
//    - a boolean indicating whether we are downloading the specified
//      infohash or just acting as a passive node
d.PeersRequest(string(infoHash), false)

// We can get the responding peer(s) by listening (or in this case iterating)
// over all the results returned by the DHT's PeersRequestResults channel
for infohash, peers := d.PeersRequestResults {
    // Do something useful with the received peers
}
```

During my testing of the library I wished at one point to run several instances of the DHT on single machine. This prompted me to go ahead and set the port number to 0, which by convention means to let the OS auto-assign a free port. This actually revealed a bug in the library.

As well documented as the simple public API of the library is, the internal working of the library were not particularly well documented or indeed commented. This made tracking down the bug challenging until it was suggested somewhere that I use a profiling tool's call graph to gain a better understanding of the relationships between the private functions. I ran the Go profiling tool against the `find_infohash_and_wait` example that is bundled with the library's code and which is essentially the above example

source Go libraries. It does this by checking their code out of their respective Version Control System (VCS) when ever a user checks the project's documentation page. This is made possible by the way the Go programming language constructs package names to include the Universal Resource Locator (URL) of the VCS system that contains the code.

[6] `https://github.com/nictuku/dht`

code wrapped in a main function. This produced the call graph in figure 4.4.

The bug came from the fact that the library did not check what port the OS returned and simply took at face value the port number the library user supplied, this meant that the library was reporting to other peers that it was reachable at port 0 which of course makes no sense. I wrote a patch, cf. appendix D.1, for the library and submitted it to Junqueira who was very encouraging and after asking me to review one change I made merged my patch into the upstream project.[7]

The specification of the Mainline DHT is that it announces the Transport Control Protocol (TCP) port at which the participating peer can be reached. The specification does not define what to do if a single IP address publishes more than one port number for a given IP address. Some libraries actually replace the port number in the existing entry. This is something I only realised during the implementation phase. This does unfortunately potentially introduce non-deterministic behaviour if there are multiple hosts sharing a single address.

4.4. Establishing a connection

As established in section 3.2, many if not most end-user's hosts are behind NAT, which must somehow be circumvented in order to enable P2P communication.

We discussed not using super-peers and therefore must map ports on routers as our only way in. Fortunately many routers support it. Unfortunately not all. This may therefore remain a theoretical exercise.

4.4.1. Using SSDP and UPnP to create port mappings in compatible routers

For the purposes of this project I developed a small Go library for controlling a UPnP enabled IGD as specified in [9].

When I was still thinking about using the C programming language, I happened upon the MiniUPnP library[8] (and MiniUPnPc UPnP port mapping client). The project includes a UPnP client to change mappings on UPnP enabled IGDs which is really quite simple. When I determined that I would write my own Go library I decided to base its API on it.

The library is called GoUPnP. It is GNU Public License Version 2 (GPLv2) licensed and available from GitHub[9]. In this section we will development process behind the library and associated binary, GoUPnPc (the *c* stands for client).

It is important first, to known the process by which a client creates a port forwarding rule on a UPnP enabled IGD:

1. The client uses the Simple Service Discovery Protocol (SSDP) [10] to discover the location and Simple Object Access Protocol (SOAP) endpoint of a UPnP enable

[7]This was very satisfying process, getting one's code merged into an upstream open-source project, I am encouraged to submit more patches to projects now.

[8]`http://miniupnp.free.fr/` or `http://miniupnp.tuxfamily.org/`

[9]`https://github.com/nhelke/goupnpc`

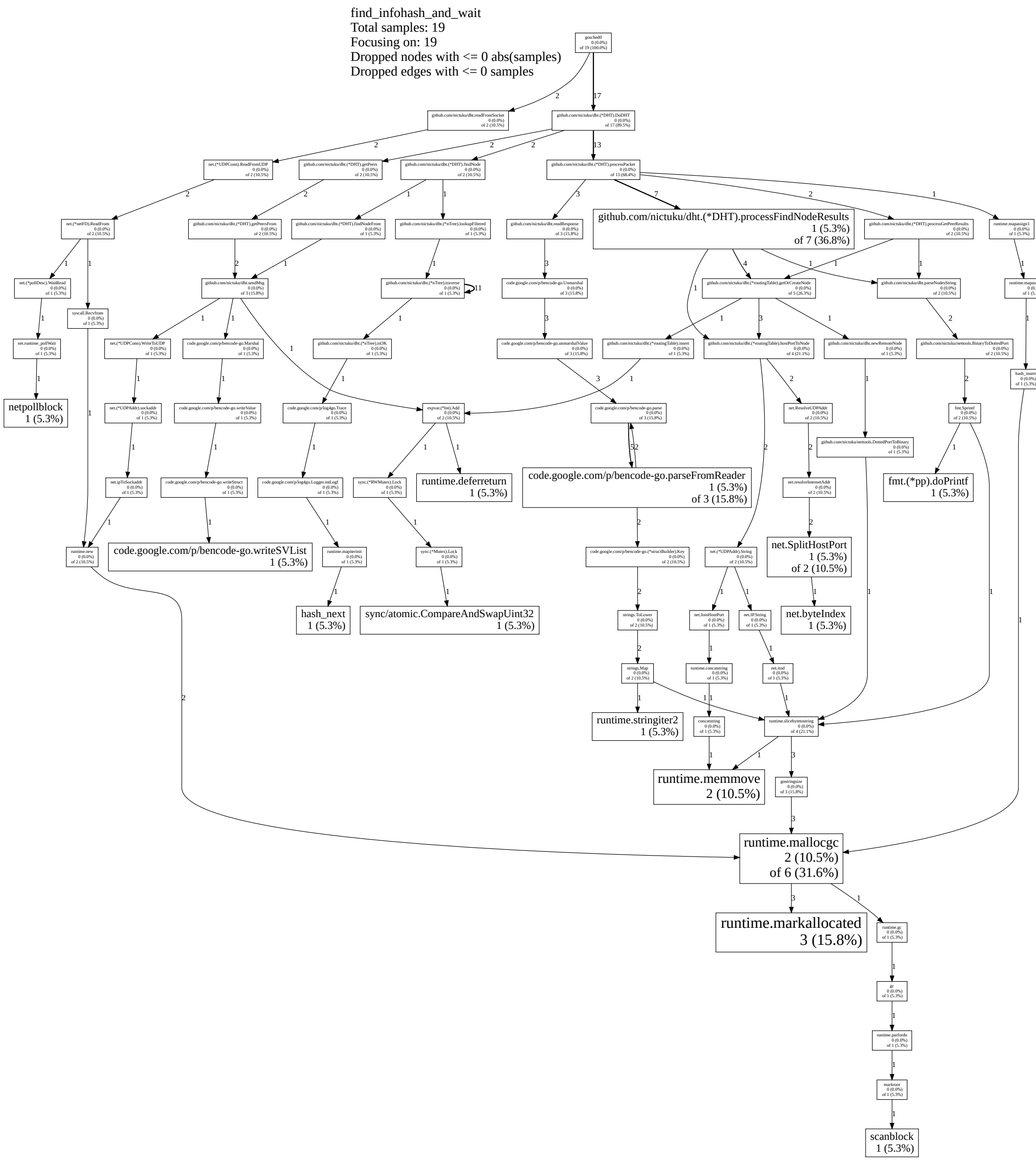

find_infohash_and_wait
Total samples: 19
Focusing on: 19
Dropped nodes with <= 0 abs(samples)
Dropped edges with <= 0 samples
github.com/nictuku/dht.(*DHT).processFindNodeResults 1 (5.3%) of 7 (36.8%)
netpollblock 1 (5.3%)
runtime.deferreturn 1 (5.3%)
code.google.com/p/bencode-go.parseFromReader 1 (5.3%) of 3 (15.8%)
fmt.(*pp).doPrintf 1 (5.3%)
code.google.com/p/bencode-go.writeSVList 1 (5.3%)
net.SplitHostPort 1 (5.3%) of 2 (10.5%)
hash_next 1 (5.3%)
sync/atomic.CompareAndSwapUint32 1 (5.3%)
net.byteIndex 1 (5.3%)
runtime.stringiter2 1 (5.3%)
runtime.memmove 2 (10.5%)
runtime.mallocgc 2 (10.5%) of 6 (31.6%)
runtime.markallocated 3 (15.8%)
scanblock 1 (5.3%)

Figure 4.4.: The call graph from profiling a run of the `find_infohash_and_wait` was useful to understand the relationships between the various undocumented private functions of the library. This is hard to see on paper, but seeing as it is a vectorial graph, those with a digital copy of this document can blow it up without losing any quality.

IGD on the Local Area Network (LAN)

   a) First the client sends a multicast HTTP Request with method `M-SEARCH` and a `ST` header indicating the type of device the client wishes to find.
   b) The client then listens on the User Datagram Protocol (UDP) socket it used to send the search for HTTP replies from available devices matching the requested type and in particular the SOAP endpoint for further UPnP interaction.

2. Once the device has been discovered, the client can control it though SOAP calls. For port mapping a single request suffices in most cases.
   a) The client request a particular port mapping from the IGD.
   b) The IGD replies with the mapping or an error
   c) If an error was returned the client may try again from 2a

Before writing my own UPnP port mapping library I did search for SSDP libraries written in Go. This search produced two results, a completely undocumented package and a reasonably well documented subpackage of *go-sonos*[10] which looked quite promising. After failing to be able to get the example code to produce any results, closer inspection of the library's code revealed:

```
func (this *ssdpDefaultManager) ssdpIncludeNotification(msg *ssdpNotifyMessage) {
    /*TODO*/
}
```

So much for documentation then! While I was looking into patching the missing parts of the library I uncovered three other issues. Its API required, for some unknown reason, that the API-user choose which port number to use for its otherwise completely internal socket. The problem was that if the port was already occupied the API didn't simply return an error, it panicked and exited the whole program. This struck me as a rather crippling race-time condition, how can I ever be sure that the port I think is free will still be free when the API gets around to creating the socket. This issue coupled with concerns I had about the maintainability of the code which was not in its own repository, led me to implement just the parts of SSDP I needed and integrate them directly into the code of GoUPnPC.

Using a network dump of the SSDP packets generated by MiniUPnPc, its code and the standard [10] I implemented just what was necessary of SSDP in order to find the IGD and nothing more.

SSDP uses HTTP over UDP for communication. One of Go's strengths is its rich standard library which includes an HTTP package. Initially when I was writing my solution I used the facilities of the standard HTTP package to modify the used network transport. However I encountered some issues with URL handling. Most notably the standard library URL package would not accept `*` as a valid URL, rightly so, however SSDP uses `*` as a path with its `M-SEARCH` verb.

[10] `https://github.com/ianr0bkny/go-sonos`

My initial reaction was to just work around this issue by overloading the URL package as well. Soon, however, my work-around code had surpassed the Source Lines of Code (SLOC) that hand crafting the UDP packet would require, so I switched techniques and in the process learned a valuable lesson.

We are told time and time again, all through are studies, that we should be lazy and reuse monolith libraries, which exists for almost anything you can imagine, and which, we are told, have been developed and tested by better men than we are.

I would tend to agree that such libraries often provide unparalleled quality and reliability as they are developed by specialists in that field who have a very good understanding of all the aspects and can therefore not only develop good backend code but a usually very well thought out specific APIs. Most of the time these specific APIs save the API-users from making mistakes, however they can get in your way when you are working on or with an edge-case.

These unforeseen edge-cases do not fit the standard. It would be wrong to modify the HTTP or any standard library to allow such a bizarre and wrong URL or other edge-case. So the correct solution, and the lesson learned, is to find a work-around which impacts the standard library in the least possible way and, importantly, does not depend on the standard library's internal behaviour as my initial work around accessing the encapsulated `RawString` field inside the standard library's URL object did.

My second attempt involved manually crafting the request, using plain old string, and sending it on a raw UDP socket, receiving the response on the raw socket and then passing that to the HTTP response parsing code from the standard library. It is a testament to the quality of the standard libraries that this was not only possible but relatively easy, in particular the accessibility of the HTTP response parsing function.

We now take a look at some actual code, corresponding to what we have just discussed. It is not the actual code used in the published version of the GoUPnP library, that can be found in function `discoverIGDDescriptionURL` in *ssdp.go* in appendix C. Instead we review some snippets based on the actual code, but cleared of some of the error checking that is done in the real code for the sake of conciseness.

First we quickly take a look at the constants used in said function and in particular we note the `format` string, which is used to hand craft the HTTP over UDP request.

```
const (
    ssdpIPv4Addr = "239.255.255.250"
    ssdpPort     = 1900
    format       = "M-SEARCH * HTTP/1.1\r\n" +
        "HOST: %s:%d\r\n" +
        "ST: %s\r\n" +
        "MAN: \"ssdp:discover\"\r\n" +
        "MX: %d\r\n" +
        "\r\n"
)
```

We recognise in the `format` string the headers of an HTTP request and we note the use of `*` as a URL.

This is an opportunity for us to introduce a Go language construct. The `const` parenthetical is just a shortcut to avoid repeating `const` on each line. It also works with `var` and `import`. For instance, instead of writing:

```
import "fmt"
import "net"
```

one could write:

```
import (
    "fmt"
    "net"
)
```

SSDP can only search for one device type at a time or all devices at once. Searching for all devices at once would potentially yield unreasonably many results as more and more household appliances connect to our LAN and are UPnP-enabled such as network-attached storage or smart televisions.

IGDs come in many different flavours. We have already excluded using the catch-all mechanism with SSDP. Unfortunately therefore, in order to find each and every available IGD, we must search for the following devices on the network, one at a time:

```
    var deviceTypes = []string{
        "urn:schemas-upnp-org:device:InternetGatewayDevice:1",
        "urn:schemas-upnp-org:service:WANIPConnection:1",
        "urn:schemas-upnp-org:service:WANPPPConnection:1",
        "upnp:rootdevice",
    }
```

The last one is a catch all, the standard shouldn't require it but as MiniUPnP uses it as a last resort I decided to add it as well.

For each successive type we try to get a response:

```
    for i := 0; i < len(deviceTypes); i++ {
        conn, err := net.ListenUDP("udp4", allIf)
        if err == nil {
            // We want to timeout and move on to the next type after a couple of
            // seconds
            conn.SetDeadline(time.Now().Add(timeout))
            // Send multicast request
            conn.WriteToUDP([]byte(fmt.Sprintf(format, ssdpIPv4Addr, ssdpPort,
                deviceTypes[i], timeout/time.Second)), broadcast)
            // Allocate a buffer for the response
            buf := make([]byte, 1500)
            for {
                // Get a response; the above timeout is still in effect as it
                // should be, so that we never wait indefinitely on any given
                // iteration
                n, addr, err := conn.ReadFromUDP(buf)
                if err != nil {
                    l4g.Info(err)
                    break
                }
                // As the http package's response parsing
                // function requires an associated request
                // object we let the http package parse our
                // request as well
                req, err := http.ReadRequest(bufio.NewReader(bytes.NewReader(
                    requestString)))
                if err != nil {
                    // Failure to parse the request represents an assertion
                    // failure as we crafted the request ourselves and have
                    // ensured its validity
                    panic(err)
                }
                // This is the magic line where we use the http
                // package to read the response
                resp, err := http.ReadResponse(bufio.NewReader(bytes.NewReader(
                    buf[:n])), req)
                // Interpret the response and break out of loop if successful
                ...
            }
        } else {
            l4g.Warn(err)
        }
    }
    // If we get here we could not find any UPnP devices
```

As every single method in the library requires at least one round-trip to the IGD, and in order to offer API-users more flexibility in the way they use the API I decided to make all the methods asynchronous. One great thing about Go is that concurrency is build directly into the language. Go uses *channels* for communication between *green-threads* which it calls *goroutines*. These channels are very similar to Ada task entry-points, both of which are based on bounded-buffers.

We will use the GetConnectionStatus method as an example:

```
// This method fetches the status of the IGD.
//
// Errors are indicated by the channel closing before a ConnectionStatus is
// returned. Listeners should therefore check at the very least for nil, better
// still for channel closure.
//
// NOTA BENE the channel closes after a successive ConnectionStatus has been
// send on it, in order to not leak resources.
func (self *IGD) GetConnectionStatus() (ret chan *ConnectionStatus) {
    // We initialise the channel
    ret = make(chan *ConnectionStatus)

    // We go do the work in a separate goroutine, the closure has access to the
14      // channel we just instanciated so we will be able to manipulate it.
15      go func() {
16          resp, ok := self.soapRequest("GetStatusInfo",
17              statusRequestStringReader(self.upnptype))
18          // In actual fact this goroutine is far more complicated because
19          // obtaining the IGD's status requires several soapRequests
20          // which all have to be successful
21          if ok {
22              // This again is a simplification, just know that ip is
23              // defined from traversing resp
24              ...
25              ret <- &ConnectionStatus{true, ip}
26              return
27          } else {
28              // Error handling
29              ...
30          }
31          // Note how if there is an error, the channel is closed without
32          // ever being send anything, upon success, the channel is only
33          // closed after the ConnectionStatus is consumed
34          close(ret)
35      }()
36
37      // We immediately return the channel to the caller
38      return
39  }
```

There are quite a few things that need to be noted about this code. First let us quickly review the method signature on line 9. Right after the keyword `func` and before the function name we see a parenthetical naming and specifying the object type that this method will act on. In go the presence of this this parameter before the function name is the difference between a function and a method that can only be called on instances. After the empty parentheses indicating that this method takes no arguments we see another parenthetical, similar to the first. This is the return declaration. We could just have indicted the type without parentheses, however naming our return value is advantageous here.

It lets us assign directly to it on line 11, where we initialise the channel, and then, as it counts as a normal variable within the scope of the function, and therefore is captured by the closure, so we can freely use it without passing it in the goroutine's parameters.

Now let us see how easy it is to start a goroutine This can be seen on line 15. This is only made easier by Go's support for anonymous functions and closures. The combination of these features is used to avoid polluting the package namespace with non-reusable functions (anonymous function) and saves us the trouble of passing the channel to the goroutine as it automatically capture the calling stack frame (closure).

On line 25 we note how objects are send down a channel with the `<-` operator. Note that this is blocking meaning that we will not reach line 36, where the channel is closed, before the object is consumed by a listener on the channel.

On the same line the `&` has the same meaning as in C and returns a reference *in lieu* of the actual struct. What follows the ampersand is merely Go's compound literal syntax. In C this would usually result in non-deterministic behaviour as you would be returning a reference to an object on a stack frame that by its very definition is about to be popped off the stack and will no longer exist after the function returns. Go's compiler has *escape analysis* which detects what objects are leaving the scope and automatically generates the code at compile time to allocate them on the heap at runtime.[11]

The documentation (lines 1–8) and the comment on lines 31–33 assume some understanding of the way channels work. It has been said before that channels are bounded-buffers. Each object send on the channel must be consumed, however if there are several listeners (or consumers) one is chosen pseudo-randomly to receive the object and all others are left in the waiting state. All waiting listeners on a channel are unconditionally released upon channel closure, thereby eliminating one possible cause of deadlocking.

We now turn to the use of this function. While there are no limits to what one could do around calling this function, there are two cases I would like to review, the synchronous and the asynchronous. The use of channels in Go makes it trivial to do either.

In order to make the function blocking or synchronous, on simply need to instantly consume the channel:

```
status := <-igd.GetConnectionStatus()
```

The `:=` is a nifty way to declare a variable and let the compiler figure out the type. The above is therefore equivalent to:

```
var status *ConnectionStatus = <-igd.GetConnectionStatus()
```

More importantly though, we must note how consuming the channel uses the same `<-` operator used to send to it, only this time the channel goes on the right of the arrow. This is actually quite intuitive.

Recall that the method returns a channel. We can also just store the reference to the channel, do something else for a bit, and consume it later, at our leisure:

```
statusChannel := igd.GetConnectionStatus()
// Do something else for a bit, the gorouting we started in GetConnectionStatus
// is working away at the same time
...
status := <-statusChannel
```

We note how using this pattern of returning a channel for long-running functions gives the API-user the choice of two very different paradigms without the overhead of having separate asynchronous and synchronous APIs.

For this reason the entire GoUPnP library uses this pattern, as you can see in the documentation available from GoDoc[12].

[11] This feature is a very good example of something that Go does much better than C. This is precisely the sort of improved language feature that I did not wish to deprive myself of by using C.

[12] `http://godoc.org/github.com/nhelke/goupnpc/goupnp`

4.4.2. On not implementing PCP *née* NAT-PMP

NAT Port Mapping Protocol (NAT-PMP) [5] or Port Control Protocol (PCP) [32] as it should henceforth be known turns out to be more difficult to implement—in Go at least. It is actually the far superior protocol, and is probably easier to implement than UPnP using C, as it doesn't use tedious and expensive-to-parse eXtensible Markup Language (XML).

In contrast to UPnP's two-step discovery-then-control process, PCP goes straight to step two and relies on the OS's routing table to determine the location of the IGD. This is far more sensible than using SSDP, and eliminates potential abuses or honest mistakes. The UPnP protocol does not check that the IGD it is controlling is actually an IGD being used by the current host, as it never checks the local routing table.

So while PCP may well be technically superior and a more efficient solution than UPnP, the problem is that there are no cross-platform system calls to inspect a kernel's routing table. A possible solution would be to let a platform dependent front-end communicate the discovered IGD to the platform independent backend, however this does not work with the premise of starting the backend as a daemon. Fortunately this problem can be tabled for the time being as UPnP is more prevalent. The only big name to support only PCP in their routers is Apple (they in fact authored the original NAT-PMP), however they are hardly a big player when it comes to the home-router market.

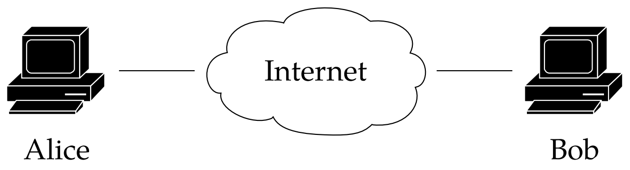
4.4.3. Confronting our hypotheses with reality

After implementing the GoUPnP library we were able to start testing hypothesis from section 3.2, i.e. that communication is possible between two peers with each other in their contact lists provided one of them is publicly addressable from the internet, either directly (rare) or through port mapped NAT (more common).

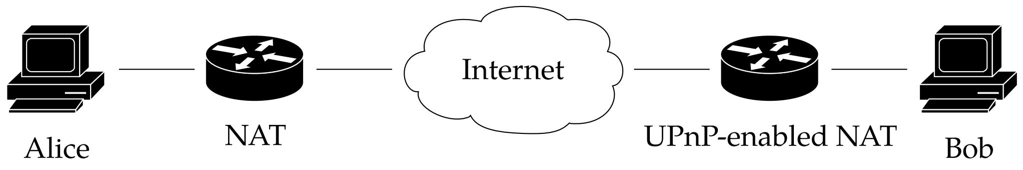
Fortunately during our testing our hypotheses were largely confirmed. We did however encounter two scenarios which we had not anticipated:

1. The so-called double-NAT. To our surprise this scenario turned up more than once, we had initially discarded it, assuming it to be of marginal importance. In a bizarre twist often the NAT closest to the client supported UPnP but only mappings to the second LAN, no control over the Wide Area Network (WAN). This case is illustrated in figure 4.5(d).

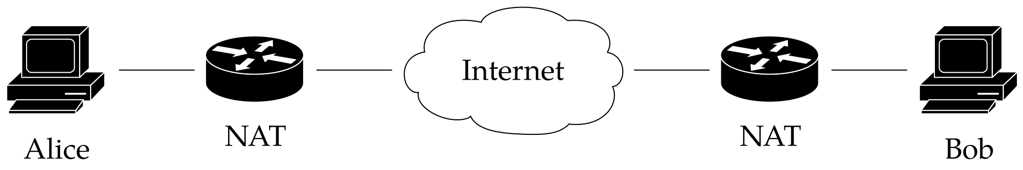
2. The second case which we had not anticipated was certain corporate and higher educational institutions have firewalls that actively detect Mainline DHT traffic (and indeed other BitTorrent traffic) and completely prevent it. This case completely surprised us. We did not expect the Mainline DHT which uses a separate protocol to BitTorrent's transfer protocol to be blocked in this manner. This case is illustrated in figure 4.5(e)



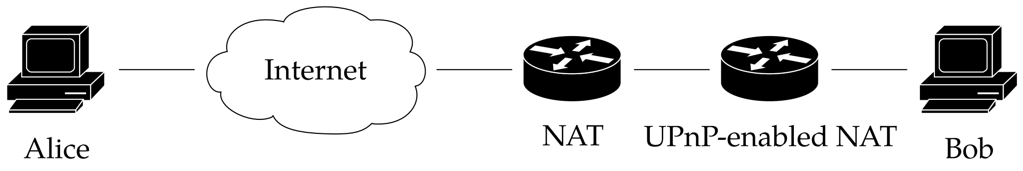
(a) Base case where Alice and Bob are both publicly addressable. Communication is possible.



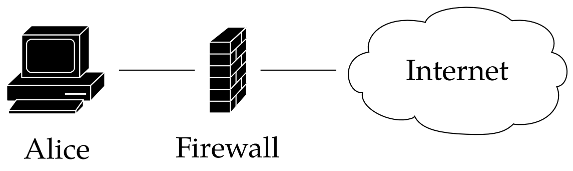
(b) Alice is behind a NAT and Bob behind a UPnP-enabled router, enabling him to open the necessary ports for Alice to contact him. Communication is possible.



(c) Alice and Bob are behind NATs. Neither NAT is UPnP-enabled. Communication is **not** possible.



(d) Alice is directly connected to the internet while and Bob is behind a double NAT where the second one is UPnP-enabled. Bob can control the UPnP-enabled IGD but this only opens ports to the second LAN. Communication is **not** possible.



(e) Alice is behind a firewall that blocks outgoing Mainline DHT packets. Communication is **not** possible. Alice doesn't know where any other peers are without access to the DHT.

Figure 4.5.: Various connection scenarios where Alice is trying to initiate a connection with Bob

4.5. Security

4.5.1. Public Key Infrastructure

As already discussed in section 3.3, Pretty Good Privacy (PGP) will be used for our Public Key Infrastructure (PKI). Fortunately there is a Go library for PGP called *openpgp*[13].

One fiasco I experienced while first trying the library was that the canonical example[14] would panic at runtime:

```
panic: crypto: requested hash function is unavailable
```

It turns out that the *openpgp* package has a bug[15]. It fails to link in a required library. In Go linking is done automatically based on the packages that are imported into the application. This failure can be fixed by manually importing the missing package:

```
import _ "code.google.com/p/go.crypto/ripemd160"
```

The underscore indicates that we do not use the package ourselves, but merely wish to link it into the program. Since Go binaries are statically linked, to avoid bloating binaries unnecessarily it is a compile error to import a package and not use it. The underscore override this on per package basis.

PGP was originally supposed to be solely the handshake procedure. Ultimately it was decided to use PGP for all peer exchanges. If you recall the theoretical plan was to use Advanced Encryption Standard (AES) once the handshake was completed. AES unfortunately was too low level. It would have required rolling our own padding and boundary detecting code. PGP packets conveniently take care of all this for us already. Very convenient for passing each message on to the protocol decoder, discussed in section 4.6. A potential draw back of PGP over AES is resource requirements. During development we did not however encounter any throughput issues with using PGP instead of AES. It is possible that this might become an issue with other higher throughput low-latency media such as voice and/or video, but these cases can be handled on an *ad hoc* basis as the protocol is designed to be extendible.

4.5.2. Adapting the handshake to symmetric NAT initiators

As discussed in section 3.2, sometimes the initiator will be behind a NAT that cannot automatically be controlled using UPnP. In this case the initiator has no reliable way of ascertaining in a decentralized manner his public IP address before the handshake. In order to handle this case while still meeting the self-imposed requirements for the handshake, the procedure must be adapted.

We note first of all that while the initiator does not always know his/her own IP address, the receiver always knows his/her IP address either because the receiver is directly connected to the internet and has a public IP or is at least behind an automatically controllable IGD and has been able to establish this information using UPnP.

[13]https://code.google.com/p/go.crypto/openpgp
[14]https://www.imperialviolet.org/2011/06/12/goopenpgp.html
[15]https://groups.google.com/d/topic/golang-nuts/BLkPi_JvMtI/discussion

```
{"name":"Nicholas Helke","email":"nhelke@gmail.com","key":138}
                              &
d4:name14:Nicholas Helke5:email16:nhelke@gmail.com3:keyi138ee
```

Figure 4.6.: A comparison of a JSON document (above) and a *bencoded* document (below) representing the same data.

Instead of supplying the receiver with our IP address we encrypt his/her address in the packet. This prevents the packet from being replayed to other hosts belonging to the same user. The handshake is now vulnerable to replay attacks within the hour targeting the same user at the same location. In order to remove this possibility, receivers must keep track of random tokens they have received and not allow the same token to be used twice within the timestamp tolerance period, i.e. one hour.

This should prevent all replay attacks, except when a malicious host intercepts the initial handshake packet and the receiver never receives it. In this case the malicious host can replay that packet from any IP address up to one hour after capture.

Another important change compared with the theoretical solution discussed in section 3.3 is not using AES with the token as a key, but instead, continuing to use PGP. This may be changed down the road, but in the mean time we are not constrained by computing resources and PGP conveniently detects individual messages and buffers them into a single readable byte slice for us.

4.6. Communication protocol

The terms of reference on page iii call for a platform-independent network protocol to be defined. This section discusses the design of a conforming network protocol. Initially as the only requirement is text communication, a basic protocol meeting those requirements is presented. The terms of reference also require that the protocol be extendible. In a later subsection the means by which the protocol might be extended are discussed and in particular a strategy to enable the use of UDP (necessary for voice and video for example) while the rest of the protocol is based on TCP.

4.6.1. Basic protocol

In order to meet the requirement of having a platform-independent protocol, and given that the DHT protocol already uses *bencoding* for serialisation, I decided to do the same.

Bencoding[6, § bencoding] is the serialisation format invented by Bram Cohen for Torrent files and since used throughout the BitTorrent eco-system, including the Mainline DHT. It is far more compact than XML and in some ways easier to parse than JavaScript Object Notation (JSON), as strings are prepended with their size, enabling the parser to dynamically allocate the correct size items and not have to worry about resizing them. It is however less human readable as we can see in figure 4.6

The theoretical advantage of *bencoding* over JSON may not actually hold up in the real world. There are far more JSON libraries than there are *bencode* ones and I would not be surprised if all that competition has produced faster JSON libraries (relative to bencode). Fortunately should we ever decide to make such a change it can be done in a backwards compatible way as all the protocol's messages are dictionaries and the starting character of a dictionary in bencode is "d" whereas JSON uses "{".

We have just revealed in passing that all the messages of the protocol are *bencoded* dictionaries. There are five different message types. The fields contained in the various dictionaries of the different message types are documented below:

SYN This is the initiating message of the handshake procedure. Its fields are:

- **token** A cryptographically secure random string
- **dest** The destination IP address in decimal-dotted format
- **timestamp** The current epoch time as an integer

SYNACK This is the response to a handshake challenge.

- **token** The token received from the initiator in it *SYN*-message
- **source** The source IP address in decimal-dotted format
- **timestamp** The current epoch time as an integer

Ping This is a ping request. It can double as an *ACK* for the handshake procedure. Its fields are:

- **token** A random token (to track the response)

Pong This is a ping response.

- **token** The token received in the *ping* request

Text Message This is a text message. Its only field is:

- **txtMsg** The message contents

4.6.2. Extending the protocol

The use of dictionaries makes the protocol easily extendible simply by ensuring that new message types have dictionary keys that differ from the above described ones. This lets anyone develop extensions to the protocol.

In order to enable organic enhancements to the protocol by third parties, we specify that clients must ignore message types they do not recognize and need not reply to them. It is the responsibility of protocol extenders to develop the message-types and message-passing diagram necessary in order to establish that a client supports a given extension. Additionally, in order to better guard ourselves against accidental message-type conflicts, protocol extenders should prepend their dictionary keys with something unique to them, such as a domain name they control.

The terms of reference (page iii) are very clear, the protocol must support arbitrary communication including but not limited to voice and video. Unfortunately the above does not work for real time voice and video as it uses TCP, whereas real time multimedia needs UDP.

Another potential advantage of UDP is that it works with full cone NATs. Full cone NATs are defined in [20] to be NATs that accept incoming UDP packets from any host to a translated address of a host behind the NAT once at least one packet has been send out, thereby establishing the translation. Additionally Saikat Guha and Paul Francis estimate in [13] 70.1'%' of the world's NATs to be full cone. This means that if we could use UDP instead of TCP even for the basic protocol and text, we would be able to achieve better penetration than the penetration discussed in section 4.4, as even routers not supporting UPnP could now serve as endpoints.

Our control protocol however is based on TCP and therefore makes the assumption that its transport is reliable. In order to use this protocol over UDP either the protocol would have to be changed to build handle retransmits or—

It turns out that Torrent clients, which suffer from similar issues when running in fully decentralised mode, have already solved this question by developing Micro Transport Protocol (µTP)[16]. µTP is a reliable transport layer build on top of UDP which provides higher level protocols with a means to send traffic reliably over UDP.

As the Mainline DHT only reports a single port number for a given IP address, µTP connections are normally accepted on the same port as DHT communication. This has the added bonus of reducing the number of ports that need to be mapped to the outside world. It does however mean that the two distinct protocols, KRPC and µTP, must be multiplexed.

The µTP protocol header (version 1) is illustrated in figure 4.7. KRPC is defined as a bencoded dictionary which means that every KRPC packet starts with a single ASCII-coded `'d'`. Figure 4.8 shows how irrespective of the type of µTP packet, the last four bits of the first byte do not match between the two protocols and will not until a version 4 is released, at which time one could easily choose to skip that value to avoid ambiguity. This means that is is indeed possible to multiplex the two protocols.

This nonetheless would require substantive modifications to the DHT library to enable multiplexing and demultiplexing of the two protocols. Two possibilities were examined:

1. The first would be to patch the DHT library to delegate network communication to the calling application. This it turns out is pretty unrealistic as the entire library was based on the idea of it handling its own socket and in particular throttling it as necessary to defend against bad nodes.

2. The second solution is to add an optional handler that the library can call if it doesn't recognize the incoming packet as starting with a `'d'`. This solution is probably the way to go as it would not only be easier to implement but also, unlike the former, it does not require that the API be changed and therefore stands a better chance of being merged into the upstream project.

[16]Originally known as µTorrent Transport Protocol in the seminal paper[24]

```
0       4       8               16              24              32
+-------+-------+---------------+---------------+---------------+
| type  | ver   | extension     | connection_id                 |
+-------+-------+---------------+---------------+---------------+
| timestamp_microseconds                                        |
+---------------+---------------+---------------+---------------+
| timestamp_difference_microseconds                             |
+---------------+---------------+---------------+---------------+
| wnd_size                                                      |
+---------------+---------------+---------------+---------------+
| seq_nr                        | ack_nr                        |
+---------------+---------------+---------------+---------------+
```

Figure 4.7.: µTP version 1 header defined in [24]

```
0        4         8
+--------+---------+
| type   | 0 0 0 1 | uTorrent transport protocol
+--------+---------+
+------------------+
| 0 1 1 0  0 1 0 0 | KRPC
+------------------+
```

Figure 4.8.: Comparision of the first 8 bits of µTP version 1 and KRPC

Extending the application to also support incoming connections when behind a full cone NAT would require either the handshake to be modified one more, as a host behind a full cone NAT does not know its public IP address and therefore cannot protect itself adequately against replay attacks. Alternatively the handshake can be left unchanged, however a means must be found for this host to establish reliably its public IP address. As we have already discussed in section 3.2 this is far from trivial to achieve in a fully decentralised manner.

In any event I think the case for the extensibility of this project's protocol has been successfully made.

5. Conclusion

My work on this thesis was divided into two parts. A first part, in parallel with classes, during which I explored almost every and any avenue remotely relevant to my subject. As is to be expected, I met with dead-ends along the way. This is the general idea of discovery phase and means that during the second part of the project, the part that is spent full time on the project, the relevant paths are known, and can be explored to greater depths while simultaneously finishing the accompanying project.

In finishing up a project, the conclusion gives one a valuable opportunity to look back and draw lessons from things done right and things done wrong. We can ponder the "what ifs" and what we might do differently, were we ever to redo the project.

The theoretical aspects of this project are quite solid I would argue. It is clearly possible in theory to develop a completely distributed communication system over Internet Protocol (IP). Additionally the solution discussed in this paper would, in theory, have helped in at least Egypt and Libya. In developing the theory however it became clear to me that this is merely one round in the constant game of cat and mouse and that it is very likely that means could be found to disrupt this theoretical solution. Perhaps the most alarming realisation was the extent to which even western governments control and censor the internet. This leads one to wonder whether the internet can be relied on at all.

As I was concluding this work, in the last week, it dawned on me how much of my work is based theory or standards or builds on elements from the BitTorrent ecosystem. This nonetheless leads me to my only theoretical "what if". What if I had known of or discovered during the course of this project other, different ecosystems that I might have built on. Bitcoin seems to be inspiring other people to come up with new decentralised systems such as Bitmessage or Namecoin, both mentioned in chapter 2. I still maintain that those systems do not lend themselves to real time communication, however they do seem to be better at traversing Network Address Translation (NAT).

On the implementation front I completed every requirement of the terms of reference. That being said my hope was always to produce widely usable solution, one which would essentially run itself and provide better overall reliability than any existing server based solutions. The discovery along the way that NAT is not nearly as easily traversed as I had hoped, was very disappointing. This means that the overall usability of the solution is severely limited.

While it is true that the solution proposed in this paper is, under certain circumstances (such as those in Egypt and Libya during the Arab Spring), more resilient than server based solutions, users will not judge it on this basis.

In photographers' circles it is oft said that "the best camera is the one you have with you." This adage perfectly summarises the issue with the solution proposed in this paper. Most users are not concerned with elegant technical solutions to rare edge-cases. Even if a solution provides unparalleled access in such cases, users are more concerned with whether a solution works day to day for them.

It does not help that the main case where my solution cannot receive connections is from behind corporate firewalls. It is in these circumstances most people spend the majority time on their computers. Telling them that this solution will work between homes if their country or region's internet was ever, all of a sudden, to become detached from the rest of the internet, is not sufficient to garner their interest.

I truly valued this opportunity to work on my own on a project. It was far from easy working on alone, particularly after my experiences working in teams on various projects all through my studies. A lesson I will have learned from this project is that it is easier to work in teams. In a team, if in doubt about a theoretical or implementation decision, you can always bounce ideas off another team member. Working alone, I had to justify every decision I made, using other people's works, to make sure that I was not going adrift. Although I also feel a certain sense of satisfaction and pride at having successfully carried this project to fruition alone.

Bibliography

[1] Egypt arrests as undersea internet cable cut off Alexandria. http://www.bbc.co.uk/news/world-middle-east-21963100, March 2013.

[2] Charles Arthur. Undersea internet cables off Egypt disrupted as navy arrests three. http://www.guardian.co.uk/technology/2013/mar/28/egypt-undersea-cable-arrests, March 2013.

[3] Scott Bradner. Key words for use in RFCs to Indicate Requirement Levels. RFC 2119 (Best Current Practice), March 1997.

[4] Dave Cheney. An introduction to cross compilation with Go. http://dave.cheney.net/2012/09/08/an-introduction-to-cross-compilation-with-go, September 2012.

[5] S. Cheshire and M. Krochmal. NAT Port Mapping Protocol (NAT-PMP). RFC 6886 (Informational), April 2013.

[6] Bram Cohen. The BitTorrent protocol specification. http://www.bittorrent.org/beps/bep_0003.html, January 2008.

[7] Scott A. Crosby and Dan S. Wallach. An analysis of BitTorrent's two Kademlia-based DHTs, 2007.

[8] Alberto Dainotti, Claudio Squarcella, Emile Aben, Kimberly C. Claffy, Marco Chiesa, Michele Russo, and Antonio Pescapé. Analysis of country-wide internet outages caused by censorship. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '11, pages 1–18, New York, NY, USA, 2011. ACM.

[9] UPnP Forum. WANIPConnection:1. Service template version 1.01, November 2001. Standardized DCP.

[10] UPnP Forum. UPnP. Device architecture version 1.1, October 2008.

[11] Dan Goodin. Skype replaces P2P supernodes with Linux boxes hosted by Microsoft. http://arstechnica.com/business/2012/05/skype-replaces-p2p-supernodes-with-linux-boxes-hosted-by-microsoft/, May 2012.

[12] Glenn Greenwald. NSA collecting phone records of millions of Verizon customers daily. http://www.guardian.co.uk/world/2013/jun/06/nsa-phone-records-verizon-court-order, June 2013.

[13] Saikat Guha and Paul Francis. Characterization and measurement of TCP traversal through NATs and firewalls. In *Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*, IMC '05, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association.

[14] D. Harkins and D. Carrel. The Internet Key Exchange (IKE). RFC 2409 (Proposed Standard), November 1998. Obsoleted by RFC 4306, updated by RFC 4109.

[15] Erik Hjelmvik and Wolfgang John. Breaking and improving protocol obfuscation. Technical Report 2010-05, Department of Computer Science and Engineering, Chalmers University of Technology, 2010. ISSN 1652-926X.

[16] Jonathan A. Kelner and Petar Maymounkov. Electric routing and concurrent flow cutting. *CoRR*, abs/0909.2859, 2009.

[17] T. Kivinen and M. Kojo. More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE). RFC 3526 (Proposed Standard), May 2003.

[18] Chris Lesniewski-Laas and M. Frans Kaashoek. Whānau: a sybil-proof distributed hash table. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 8–8, Berkeley, CA, USA, 2010. USENIX Association.

[19] Andrew Loewenstern. BitTorrent enhancement proposal #5: DHT protocol. `http://www.bittorrent.org/beps/bep_0005.html`, January 2008.

[20] R. Mahy, P. Matthews, and J. Rosenberg. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). RFC 5766 (Proposed Standard), April 2010.

[21] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 53–65, London, UK, UK, 2002. Springer-Verlag.

[22] Declan McCullagh. NSA docs boast: Now we can wiretap Skype video calls. `http://news.cnet.com/8301-13578_3-57593339-38/`, July 2013.

[23] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, May 2009.

[24] Arvid Norberg. BitTorrent enhancement proposal #29: uTorrent transport protocol. `http://www.bittorrent.org/beps/bep_0029.html`, June 2009.

[25] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '01, pages 161–172, New York, NY, USA, 2001. ACM.

[26] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing. Session Traversal Utilities for NAT (STUN). RFC 5389 (Proposed Standard), October 2008.

[27] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, pages 329–350, London, UK, UK, 2001. Springer-Verlag.

[28] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. Technical Report UW-CSE-01-06-02, University of Washington, July 2001.

[29] R. Schollmeier. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In *Peer-to-Peer Computing, 2001. Proceedings. First International Conference on*, pages 101–102, 2001.

[30] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. ACM.

[31] Jonathan Warren. Bitmessage: A peer-to-peer message authentication and delivery system. `http://www.bitmessage.org`, November 2012.

[32] D. Wing, S. Cheshire, M. Boucadair, R. Penno, and P. Selkirk. Port Control Protocol (PCP). RFC 6887 (Proposed Standard), April 2013.

[33] Charles V. Wright, Lucas Ballard, Scott E. Coull, Fabian Monrose, and Gerald M. Masson. Uncovering spoken phrases in encrypted voice over IP conversations. *ACM Trans. Inf. Syst. Secur.*, 13(4):35:1–35:30, December 2010.

[34] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22:41–53, 2004.

Acronyms

µTP Micro Transport Protocol. 44

AES Advanced Encryption Standard. 19, 41, 42

API Application Programming Interface. 24, 30, 31, 33, 34, 36, 38, 44

CUPS Common Unix Printing System. 27

DDoS Distributed Denial of Service. 8, 11, 14, 15

DHT Distributed Hash Table. i, 8–10, 15, 16, 18, 20, 23, 27, 30, 31, 39, 42, 44

DNS Domain Name System. 4, 14

EFF Electronic Frontier Foundation. 4

GPLv2 GNU Public License Version 2. 31, 51

GUI Graphical User Interface. iv, 27

HTML Hyper-Text Markup Language. 27

HTTP Hyper-Text Transport Protocol. 27, 33, 34

ICANN Internet Corporation for Assigned Names and Numbers. 4

IGD Internet Gateway Device. 14, 23, 31, 33, 35, 36, 39, 41

IP Internet Protocol. i, iii, 1, 4, 7, 14–16, 18, 19, 23, 31, 41–45, 47

IPv4 Internet Protocol Version 4. iii, 14

IPv6 Internet Protocol Version 6. 14

ISP Internet Service Provider. 1, 16, 17

JS JavaScript. 27

JSON JavaScript Object Notation. 42, 43

JVM Java Virtual Machine. 26

LAN Local Area Network. 33, 35, 39

MSE Message Stream Encryption. 17

NAT Network Address Translation. iii, 7, 14, 23, 31, 39, 41, 44, 45, 47

NAT-PMP NAT Port Mapping Protocol. 39

NDK Native Development Kit. 26

NIC Network Information Centre. 4

NSA National Security Agency. 16

OO Object Oriented. 28

OS Operating System. 27–31, 39

P2P Peer-to-Peer. iii, 1, 3, 7–9, 12, 14–16, 21, 24, 31

PCP Port Control Protocol. 14, 39

PGP Pretty Good Privacy. 20, 41, 42

PKI Public Key Infrastructure. 41

RFC Request for Comments. 17

RPC Remote Procedure Call. 13

SLOC Source Lines of Code. 34

SOAP Simple Object Access Protocol. 31, 33

SSDP Simple Service Discovery Protocol. 31, 33, 35, 39

STUN Session Traversal Utilities for NAT. 14, 15

TCP Transport Control Protocol. 31, 42, 44

TURN Traversal Using Relays around NAT. 15

UDP User Datagram Protocol. 12, 33, 34, 42, 44

UPnP Universal Plug and Play. 14, 23, 31, 33, 35, 39, 41, 44

URL Universal Resource Locator. 30, 33, 34

UTC Coordinated Universal Time. 18

VCS	Version Control System. 30
VoIP	Voice over IP. 14, 15, 17
WAN	Wide Area Network. 39
WoT	Web of Trust. 19
XML	eXtensible Markup Language. 39, 42

Appendix A.

Dictator Breaker User Manual

A.1. How to run

You run Dictator Breaker by calling its binary from the terminal (on some platforms it may also be possible to double-click the binary and a terminal will launch automatically).

The program will spit out some output among which a URL where the user interface can be found and which must be viewed in order to proceed.

Once the interface has been loaded in the browser, you will be prompted to enter the email address and passphrase of the private key from your local GPG keyring. Once you have done this, you should be able to follow in the terminal as the backend attempts to connect to your GPG contacts who are also using Dictator Breaker.

New connections and messages should appear in your browser window.

In order to send a message to an active connection prefix it with the connection identifier reported on the webpage when the connection was initialised.

Close the program either by closing the browser window or by closing the terminal.

A.2. Command line flags

```
  -cleanupPeriod=15m0s: How often to ping nodes in the network to see if they
are reachable.
  -http=8080: port from which to serve the HTML/JS GUI on the loopback interface
  -maxNodes=500: Maximum number of nodes to store in the routing table, in
memory. This is the primary configuration for how noisy or aggressive this node
should be. When the node starts, it will try to reach maxNodes/2 as quick as
possible, to form a healthy routing table.
  -port=0: use this port for the DHT and communication (UDP and TCP), 0 means
let the operating system choose a random available port
  -rateLimit=100: Maximum packets per second to be processed. Beyond this limit
they are silently dropped. Set to -1 to disable rate limiting.
  -routers="1.a.magnets.im:6881,router.utorrent.com:6881": Comma separated
IP:Port address of the DHT routeirs used to bootstrap the DHT network.
  -savePeriod=5m0s: How often to save the routing table to disk.
  -storeDHT=true: whether to store and load the DHT routing table from/to disk
```

Appendix B. Dictator Breaker Code

For archival and review purposes, the source of Dictator Breaker is included below as it stood at submission time:

B.1. go.nhelke.com/dictator-breaker/main.go

```
package main

import (
    "code.google.com/p/go.crypto/openpgp"
    l4g "code.google.com/p/log4go"
    "flag"
    "fmt"
    "github.com/nhelke/dht"
    "github.com/nhelke/goupnpc/goupnp"
    "go.nhelke.com/dictator-breaker/messaging"
    "go.nhelke.com/dictator-breaker/security"
    "math/rand"
    "net"
    "net/http"
    "os"
    "strings"
    "time"
)

const (
    targetNumPeers = 60
)

var (
    port = flag.Int("port", 0,
        "use this port for the DHT and communication (UDP and TCP), "+
            "0 means let the operating system choose a random available port")
    storeDHTRoutingTable = flag.Bool("storeDHT", true,
        "whether to store and load the DHT routing table from/to disk")
    interfacePort = flag.Int("http", 8080,
        "port from which to serve the HTML/JS GUI on the loopback interface")

    connections   = make([]*conn, 0, 200)
    newConnection = make(chan *conn)
    expConnection = make(chan *conn)
    newIncMessage = make(chan messaging.Message)
    newOutMessage = make(chan messaging.Message)
    me            string
    ws            *wsConn
)

type conn struct {
    net     net.Conn
    contact *openpgp.Entity
    in      []string
    out     []string
    send    chan map[string]interface{}
    closed  chan struct{}
}

const (
    txtMessageKey = "txtMsg"
    pingKey       = "ping"
    pongKey       = "pong"
)

// This function should be called the thread that handled the new connection,
// i.e. either accept's or dial's thread, and it only returns after the
// connection has dropped. It manages its its addition and removal from the
// connection slice itself.
func (c conn) Run() {
    c.send = make(chan map[string]interface{})
    c.closed = make(chan struct{})

    newConnection <- &c
    defer func() { expConnection <- &c }()

    go func() {
        for {
            select {
            case pack := <-c.send:
                err := security.EncryptBencode(c.net, c.contact, pack)
                if err != nil {
                    l4g.Warn(err)
                    close(c.closed)
                    return
                }

            case _, open := <-c.closed:
                if !open {
                    c.net.Close()
                    return
                }
            }
        }
    }()

    for {
        pack, signer, err := security.DecryptBencode(c.net)
        if err != nil {
            l4g.Warn(err)
            close(c.closed)
            return
        } else if signer != c.contact {
            l4g.Warn("Unexpected signer (%v) on connection %v@%v", signer,
                c.contact, c.net.RemoteAddr())
            close(c.closed)
            return
        }

        if msg, ok := pack[txtMessageKey]; ok {
            c.in = append(c.in, msg.(string))
            ws.out <- fmt.Sprintf("%s says > %s", signer, msg)
        }
        if ping, ok := pack[pingKey]; ok {
            pack := map[string]interface{}{
                pongKey: ping,
            }
            c.send <- pack
        }
        if _, ok := pack[pongKey]; ok {
            l4g.Warn("Sucessfully pinged %v@%v", c.contact, c.net.RemoteAddr())
        }
    }
}

func (c conn) SendMessage(msg string) {
    pack := map[string]interface{}{
        txtMessageKey: msg,
    }
    c.send <- pack
}

func (c conn) Ping() {
    pack := map[string]interface{}{
        txtMessageKey: security.RandomToken(),
    }
    c.send <- pack
}

```

```go
func main() {
    // Remove this line in production
    l4g.NewDefaultLogger(l4g.DEBUG)

    flag.Parse()
    if len(flag.Args()) > 0 {
        flag.PrintDefaults()
        os.Exit(1)
    }

    igd := <-goupnp.DiscoverIGD()
    if igd == nil {
        l4g.Warn("No UPnP IGD found")
    } else {
        if *port == 0 {
            *port = rand.Intn(65536-1024) + 1024
        }
        l4g.Warn("Using port %v", *port)
        tcpPortMap := <-igd.AddLocalPortRedirection(uint16(*port), goupnp.TCP)
        udpPortMap := <-igd.AddLocalPortRedirection(uint16(*port), goupnp.UDP)
        if tcpPortMap == nil || udpPortMap == nil {
            l4g.Error("Unable to map ports")
        }
        status := <-igd.GetConnectionStatus()
        if status != nil {
            me = status.IP.String()
        }
    }

    httpPort := fmt.Sprintf(":%d", *interfacePort)
    go http.ListenAndServe(httpPort, nil)
    l4g.Warn("Listening on http://localhost%s", httpPort)

    for {
        connChan := make(chan *wsConn)
        connCC <- connChan

        ws = <-connChan

        ws.out <- "GPG Email:"
        user := <-ws.in
        ws.out <- "GPG Password:"
        pass := <-ws.in

        err := security.ReadKeyring(user, pass)
        if err != nil {
            l4g.Warn(err)
            ws.ws.Close()
            continue
        } else {
            break
        }
    }

    d, err := dht.NewDHTNode(*port, targetNumPeers, *storeDHTRoutingTable)
    if err != nil {
        l4g.Critical(err)
        os.Exit(1)
    }

    tick := time.Tick(1 * time.Minute)
    for {
        select {
        case infoHashPeers := <-d.PeersRequestResults:
            for ih, peers := range infoHashPeers {
                if len(peers) > 0 {
                    for _, peer := range peers {
                        addr := dht.DecodePeerAddress(peer)
                        l4g.Debug("peer found for infohash %x@%v", addr, ih)
                        dial(string(ih), addr)
                    }
                }
            }
        case newConnection := <-newConnection:
            connections = append(connections, newConnection)
        case expConnection := <-expConnection:
            for i := 0; i < len(connections); i++ {
                if connections[i] == expConnection {
                    connections[i] = connections[len(connections)-1]
                    connections = connections[0 : len(connections)-1]
                    break
                }
            }
        case <-tick:
            for _, c := range security.Contacts() {
                ih := security.FingerprintToString(c.PrimaryKey.Fingerprint)
                d.PeersRequest(ih, me != "" && ih == security.MyFingerprint())
            }
        }
    }
}

// This should be called on a separate thread
func listen(port int) {
    ln, err := net.Listen("tcp", fmt.Sprintf(":%d", port))
    if err != nil {
        l4g.Critical("Failed to open TCP socket on same port as UDP", err)
        os.Exit(5)
    }
    for {
        conn, err := ln.Accept()
        if err != nil {
            l4g.Error(err)
            continue
        }
        go accept(conn)
    }
}

// This should be called on a separate thread
func accept(c net.Conn) {
    l4g.Debug("Accepted connection from %v", c.RemoteAddr())

    reply, signer, err := security.DecryptBencode(c)
    if err != nil {
        l4g.Warn(err)
        c.Close()
        return
    }

    token := reply["token"]

    if me != reply["dest"] ||
        !security.WithinOneHour(reply["timestamp"].(int)) {
        l4g.Warn(err)
        c.Close()
        return
    }

    synack := map[string]interface{}{
        "token":     token,
        "timestamp": time.Now().Unix(),
        "source":    me,
    }

    err = security.EncryptBencode(c, signer, synack)
    if err != nil {
        l4g.Warn(err)
        c.Close()
        return
    }

    conn{
        net:     c,
        contact: signer,
    }.Run()
}
```

```

// This should be called on a separate thread
func dial(ih, addr string) {
    conn, err := net.Dial("tcp", addr)
    if err != nil {
        l4g.Debug(err)
        conn.Close()
        return
    }

    token := security.RandomToken()
    remote := strings.Split(addr, ":")[0]
    syn := map[string]interface{}{
        "token":     token,
        "dest":      remote,
        "timestamp": time.Now().Unix(),
    }

    err = security.EncryptBencode(conn, security.GetKeyByFingerprint(ih), syn)
    if err != nil {
        l4g.Warn(err)
        conn.Close()
        return
    }

    synack, signer, err := security.DecryptBencode(conn)
    if err != nil {
        l4g.Warn(err)
        conn.Close()
        return
    }

    if ih != security.FingerprintToString(signer.PrimaryKey.Fingerprint) ||
        synack["token"] != token ||
        synack["source"] != remote ||
        !security.WithinOneHour(synack["timestamp"].(int)) {
        l4g.Warn("Bad SYNACK: %v", synack)
    }

}
```

B.2. go.nhelke.com/dictator-breaker/ws.go

```
package main

import (
    "fmt"
    "net/http"

    "code.google.com/p/go.net/websocket"
    l4g "code.google.com/p/log4go"
)

var (
    connCC     = make(chan chan *wsConn)
    incomingWs = make(chan string)
)

type userPass struct {
    user, pass string
}

type wsConn struct {
    ws  *websocket.Conn
    in  <-chan string
    out chan<- string
}

func APISocket(ws *websocket.Conn) {
    select {
    case connChan := <-connCC:
        l4g.Debug("Accepting WebSocket connection from %v", ws.RemoteAddr())

        var (
            in  = make(chan string)
            out = make(chan string)
        )

        go func() {
            for {
                var line string
                _, err := fmt.Fscanln(ws, &line)
                if err != nil {
                    l4g.Error(err)
                    break
                }
                in <- line
            }
        }()

        connChan <- &wsConn{
            ws:  ws,
            in:  in,
            out: out,
        }

        for {
            select {
            case str := <-out:
                fmt.Fprintln(ws, str)
            }
        }
    default:
        fmt.Fprintln(ws, "There is already an active session connected to this backend")
        ws.Close()
    }
}

func init() {
    http.Handle("/api", websocket.Handler(APISocket))
    http.Handle("/", http.FileServer(http.Dir("views")))
}
```

B.3. go.nhelke.com/dictator-breaker/messaging/message.go

```
package messaging

import (
    "encoding/json"
    "go.nhelke.com/dictator-breaker/security"
    "io"
    "net"
)

type Message struct {
    conn *net.Conn `json:"-"`

    Type  string
    Token string

    Body map[string]interface{} `json:",omitempty"`
}

func (m *Message) WriteTo(w io.Writer) (err error) {
    b, err := json.Marshal(m)
    if err == nil {
        _, err = w.Write(b)
    }
    return
}

func Ping() (m Message) {
    m.Type = "ping"
    m.Token = security.RandomToken()
    return
```

```go
}

func TextMessage(text string) (m Message) {
    m.Type = "textMessage"
    m.Token = security.RandomToken()
    m.Body = make(map[string]interface{})
    m.Body["txt"] = text
    return
}

func (m *Message) Ack() (ack Message) {
    ack = *m
    ack.Type = "ack"
    ack.Body = nil
    return
}
```

B.4. go.nhelke.com/dictator-breaker/messaging/message_test.go

```go
package messaging

import (
    "bytes"
    "testing"
)

func TestMessageMarshalling(t *testing.T) {
    msg := Ping()
    b := new(bytes.Buffer)
    msg.WriteTo(b)
    str := string(b.Bytes())
    t.Log(str)
    if str != `{"Type":"ping","Token":"4"}` {
        t.Fail()
    }
}
```

B.5. go.nhelke.com/dictator-breaker/security/openpgp.go

```go
package security

import (
    bencode "code.google.com/p/bencode-go"
    "code.google.com/p/go.crypto/openpgp"
    l4g "code.google.com/p/log4go"
    "crypto/rand"
    "errors"
    "io"
    "os"
    "path"
    "strings"
    "time"
    // This is needed by openpgp, failure to link it into the binary
    // results in cryptic panics at runtime
    _ "code.google.com/p/go.crypto/ripemd160"
)

var (
    alice   *openpgp.Entity
    pubring openpgp.EntityList
)

func ReadKeyring(email, password string) error {
    secringFile, err := os.Open(GPGFolderPath("secring.gpg"))
    if err != nil {
        return err
    }

    secring, err := openpgp.ReadKeyRing(secringFile)
    if err != nil {
        return err
    }

    alice = getKeyByEmail(secring, email)
    if alice == nil {
        return errors.New("Requested private key not found")
    }

    err = alice.PrivateKey.Decrypt([]byte(password))
    if err != nil {
        return err
    }

    pubringFile, err := os.Open(GPGFolderPath("pubring.gpg"))
    if err != nil {
        return err
    }

    pubring, err = openpgp.ReadKeyRing(pubringFile)
    if err != nil {
        return err
    }

    pubring = append(pubring, alice)

    return nil
```

```
}

func GPGFolderPath(filename string) (dir string) {
    env := os.Environ()
    for _, e := range env {
        if strings.HasPrefix(e, "HOME=") {
            dir = strings.SplitN(e, "=", 2)[1]
            dir = path.Join(dir, ".gnupg")
            path.Join(dir, filename)
            break
        }
    }
    return
}

// Shamelessly taken from the seemingly canonical OpenPGP example for Go at
// https://www.imperialviolet.org/2011/06/12/goopenpgp.html
func getKeyByEmail(keyring openpgp.EntityList, email string) *openpgp.Entity {
    for _, entity := range keyring {
        for _, ident := range entity.Identities {
            if ident.UserId.Email == email {
                return entity
            }
        }
    }

    return nil
}

func GetKeyByFingerprint(fingerprint string) *openpgp.Entity {
    return getKeyByFingerprint(pubring, fingerprint)
}

func getKeyByFingerprint(keyring openpgp.EntityList, fingerprint string) *openpgp.Entity {
    for _, entity := range keyring {
        if FingerprintToString(entity.PrimaryKey.Fingerprint) == fingerprint {
            return entity
        }
    }

    return nil
}

func Contacts() openpgp.EntityList {
    return pubring
}

func MyFingerprint() string {
    return FingerprintToString(alice.PrimaryKey.Fingerprint)
```

```go
}

func FingerprintToString(fingerprint [20]byte) string {
    return string(fingerprint[:])
}

func EncryptBencode(conn io.Writer, bob *openpgp.Entity, msg map[string]interface{}) (err error) {
    plain, err := openpgp.Encrypt(conn, openpgp.EntityList{bob}, alice, nil, nil)
    if err != nil {
        l4g.Warn(err)
    }
    err = bencode.Marshal(plain, msg)
    if err != nil {
        l4g.Warn(err)
    }
    plain.Close()

    return
}

func DecryptBencode(conn io.Reader) (res map[string]interface{},
    signer *openpgp.Entity, err error) {
    msg, err := openpgp.ReadMessage(conn, pubring, nil, nil)
    if err != nil {
        l4g.Warn(err)
        return
    }

    if !msg.IsEncrypted || !msg.IsSigned {
        err = errors.New("Message was not encrypted or not signed")
        return
    }

    signer = msg.SignedBy.Entity

    err = bencode.Unmarshal(msg.UnverifiedBody, res)
    if err != nil {
        l4g.Warn(err)
        return
    }

    if msg.SignatureError != nil {
        err = errors.New("MESSAGE SIGNATURE WRONG! MESSAGE TAMPERING DETECTED!")
    }
    return
}

func RandomToken() string {
    b := make([]byte, 32)
    n, err := io.ReadFull(rand.Reader, b)
    if n != len(b) || err != nil {
        panic("Programming error, there is no reason for this to fail")
    }
    return string(b)
}

func WithinOneHour(epoch int) bool {
    now := time.Now()
    oneHourFromNow := now.Add(1 * time.Hour)
    oneHourAgo := now.Add(-1 * time.Hour)
    x := time.Unix(int64(epoch), 0)
    return oneHourFromNow.After(x) && oneHourAgo.Before(x)
}
```

B.6. go.nhelke.com/dictator-breaker/security/openpgp_test.go

```go
package security

import (
    "bytes"
    "fmt"
    "io/ioutil"
    "testing"

    "code.google.com/p/go.crypto/openpgp"
    _ "code.google.com/p/go.crypto/ripemd160"
)

func TestHandshake(t *testing.T) {
    alice, err := openpgp.NewEntity("Alice", "", "alice@example.com", nil)
    if err != nil {
        t.Error(err)
    }
    bob, err := openpgp.NewEntity("Bob", "", "bob@example.com", nil)
    if err != nil {
        t.Error(err)
    }

    buf := new(bytes.Buffer)

    plain, err := openpgp.Encrypt(buf, openpgp.EntityList{bob}, alice, nil, nil)
    if err != nil {
        t.Error(err)
    }

    handshake := "Whatever"
    fmt.Fprintf(plain, handshake)

    plain.Close()

    msg, err := openpgp.ReadMessage(buf, openpgp.EntityList{alice, bob}, nil, nil)
    if err != nil {
        t.Error(err)
    }

    if !msg.IsEncrypted || !msg.IsSigned {
        t.Fail()
    }

    res, err := ioutil.ReadAll(msg.UnverifiedBody)
    if err != nil {
        t.Error(err)
    }

    if msg.SignatureError != nil || string(res) != handshake {
        t.Fail()
    }

}
```

B.7. go.nhelke.com/dictator-breaker/views/chat.html

```
<!DOCTYPE html>

<head>
<meta charset="utf-8" />

<title>WebSocket Test</title>

<script src="https://ajax.googleapis.com/ajax/libs/jquery/2.0.0/jquery.min.js"></script>
<script src="/jquery-2.0.0.js"></script>

<style>
@import url(http://fonts.googleapis.com/css?family=Inconsolata:400,700);
* {
font-family: 'Inconsolata', monospace;
}
#input {
    border: thin solid black;
}
</style>

</head>
<body>
<h2>Dictator Breaker</h2>
```

```

<div id="output"></div>
<div id="input" contenteditable="true"></div>

<script>

  var wsUri = "ws://localhost:8080/api";
  var output;

  function onOpen(evt)
  {
        var msg = $('<div>WebSocket connection opened.</div>');
        msg.hide()
        msg.appendTo($("#output"));
        msg.slideDown();
  }

  function onClose(evt)
  {
    alert("DISCONNECTED");
  }

  function onMessage(evt)
  {
        var msg = $('<div class="incoming">' + evt.data + '</div>');
        msg.hide()
        msg.appendTo($("#output"));
        msg.slideDown();
  }

  function onError(evt)
  {
    alert(evt.data);
  }

  $(function () {
    websocket = new WebSocket(wsUri);
    websocket.onopen = function(evt) { onOpen(evt) };
    websocket.onclose = function(evt) { onClose(evt) };
    websocket.onmessage = function(evt) { onMessage(evt) };
    websocket.onerror = function(evt) { onError(evt) };

    var input = jQuery("#input");
    var output = $("#output");
    input.focus();
    input.keypress(function(e) {
      if(e.which == 13) {
            var msg = input.clone().removeAttr("id").removeAttr("contenteditable");
            websocket.send(msg[0].innerHTML + "\n");
```

```
                msg.hide()
                msg.addClass("outgoing");
                msg.appendTo(output);
                msg.slideDown();
                input.empty();
            return false;
        }
      });
    });
</script>

</body>

</html>


```

B.8. Known issues

1. The resources for the user interface must be placed in a subdirectory of the current working directory, called "views".

Appendix C.

GoUPNPc

GoUPnPc is a GNU Public License Version 2 (GPLv2) licensed[1] library for controlling port mappings of Universal Plug and Play (UPnP)-enabled Internet Gateway Devices (IGDs). Its canonical source is GitHub[2]. Its documentation is available from GoDoc[3].

For archival and review purposes, the source of GoUPnPc is included below as it stood at submission time:

C.1. github.com/nhelke/goupnpc/README.mdown

```
GoUPnPC
=======

GoUPnPC is a MiniUPnPC inspired library I am in the process of developping as
part of my Bachelor's degree thesis. For the time being only IGDv1 is supported.
It offers an API very close to the options provided by the UPnPC command line
client.

Documentation is available at <http://godoc.org/github.com/nhelke/goupnpc>

License
-------

    Copyright (C) 2013  Nicholas Helke

    This program is free software; you can redistribute it and/or
    modify it under the terms of the GNU General Public License
    as published by the Free Software Foundation; either version 2
    of the License, or (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.
```

[1]As required by the license, a copy is available as appendix E
[2]`https://github.com/nhelke/goupnpc`
[3]`http://godoc.org/github.com/nhelke/goupnpc`

C.2. github.com/nhelke/goupnpc/cmd.go

```
// This command is useful both to test the associated goupnp library and
// its source serves as an example of how to use said library.
//
// Usage instructions can be obtained by running it without any arguments.
package main

import (
    "fmt"
    "os"
    "strconv"
    "time"

    l4g "code.google.com/p/log4go"
    "github.com/nhelke/goupnpc/goupnp"
)

func main() {
    l4g.AddFilter("stdout", l4g.WARNING, l4g.NewConsoleLogWriter())

    if len(os.Args) < 2 {
        printUsage()
    } else {
        discover := goupnp.DiscoverIGD()
        if os.Args[1] == "s" {
            igd := <-discover
            status := <-igd.GetConnectionStatus()
            fmt.Printf("%+v\n", status)
        } else if os.Args[1] == "l" {
            igd := <-discover
            for portMapping := range igd.ListRedirections() {
                fmt.Println(portMapping)
            }
        } else if os.Args[1] == "a" {
            igd := <-discover
            port, _ := strconv.Atoi(os.Args[2])
            proto := goupnp.ParseProtocol(os.Args[3])
            myMapping := <-igd.AddLocalPortRedirection(uint16(port), proto)
            fmt.Printf("%+v\n", myMapping)
        } else {
            printUsage()
        }
    }

    time.Sleep(1 * time.Second)
}

func printUsage() {
    fmt.Println(
        `Usage: goupnpc s
            Print IGD Status
        goupnpc a port protocol
            Add local port mapping with internal and external ports equal to
            port and protocol equal to, well I will let you guess
        goupnpc l
            Lists all port mappings on the IGD
NOTA BENE No error checking is performed, if anything goes wrong, it will
probably panic on nil or something`)
}
```

C.3. github.com/nhelke/goupnpc/goupnp/goupnp.go

```go
// A small library for using the port mapping controls of UPnP-enabled IGDs
package goupnp

import (
    "errors"
    "fmt"
    "io/ioutil"
    "net"
    "net/http"
    "net/url"
    "strings"

    l4g "code.google.com/p/log4go"
)

const (
    TCP protocol = 1 << iota
    UDP
)

// This type provides all the information about port mappings.
// It also serves as a handle returned by AddLocalPortRedirection() for use with
// DeletePortRedirection().
type PortMapping struct {
    InternalPort uint16
    ExternalPort uint16
    Protocol     protocol
    InternalHost net.IP
    Description  string
    Enabled      bool
    Lease        uint
}

func (self *PortMapping) String() string {
    return fmt.Sprint(self.InternalHost, ":", self.InternalPort, "<=",
        self.ExternalPort, self.Protocol, ` "`, self.Description, `" (`,
        self.Enabled, ", ", self.Lease, ")")
}

// This opaque type provides a handle to a discovered IGD
// Use DiscoverIGD() to obtain such a handle.
//
// NOTA BENE Using instances of this struct not retured by the appropriate
// function call has undefined behaviour
type IGD struct {
    controlURL *url.URL
    upnptype   string
    iface      net.IP
}

func (self *IGD) String() string {
    return self.controlURL.String()
}

// This function returns a channel which will be sent the first IGD it finds in
// traversing `net.InterfaceAddrs()` with IP addresses in the private network
// range.
//
// The channel this function returns should be listened on to avoid leaking
// goroutines. Additionally the listener must check whether the channel the
// value returned by the channel against nil, to ensure that an IGD was indeed
// found.
func DiscoverIGD() (ret chan *IGD) {
    // Create the channel we will return
    ret = make(chan *IGD)

    // Do the work asynchronously
    go func() {
        // For each and every local address in the private network range
        bindLocalAddrs := localPrivateAddrs()
        l4g.Debug("Found %d private network interfaces", len(bindLocalAddrs))
        for i := 0; i < len(bindLocalAddrs); i++ {
            // Use SSDP to search for a UPnP-enabled IGD
            descURL, ok := discoverIGDDescriptionURL(bindLocalAddrs[i])

            if ok {
                // If we found one, we go fetch its description XML
                resp, err := http.Get(descURL.String())
                if err == nil {
                    // We got something back, lets not leak it
                    defer resp.Body.Close()
                    // We read in the whole description into memory We might
                    // envisage at a later date putting an upperbound on the
                    // buffer, however there is no risk of buffer overflow, so
                    // it is a low priority
                    body, err := ioutil.ReadAll(resp.Body)
                    if err == nil {
                        l4g.Debug("Description XML:\n%s", string(body))
                        // Parse the XML and extract relevant information
                        upnptype, controlURL, err := getConnectionControlURL(body)
                        if err == nil {
                            var igd IGD
                            // It worked, lets now try and wrap it in an igd struct
                            igd.controlURL, err = url.Parse(controlURL)
                            if err != nil {
                                l4g.Warn("Failed to parse URL %v", controlURL)
                            } else {
                                // The URL was good, lets track the type as
                                // well, in order to make the correct calls down
                                // the line
                                igd.upnptype = upnptype
                                // We now add the local binding address to
                                // enable the simple AddLocalPortRedirection
                                // method
                                igd.iface = bindLocalAddrs[i].IP

                                ret <- &igd
                            }
                        } else {
                            l4g.Warn("Bad XML: %v", err)
                        }
                    } else {
                        l4g.Warn("Error reading response")
                    }
                }
            }
        }

        // If we get here we did not find an IGD or have already passed the
        // information to the channel and it has been read, so we close the
        // channel This will have the effect of returning nil and will indicate
        // the closure to listeners.
        close(ret)
    }()
    return
}

```

```
type ConnectionStatus struct {
    Connected bool
    IP        net.IP
}

// This method fetches the status of the IGD.
//
// Errors are indicated by the channel closing before a ConnectionStatus is
// returned. Listeners should therefore check at the very least for nil, better
// still for channel closure.
//
// NOTA BENE the channel closes after a successive ConnectionStatus has been
// send on it, in order to not leak resources.
func (self *IGD) GetConnectionStatus() (ret chan *ConnectionStatus) {
    // We initialise the channel
    ret = make(chan *ConnectionStatus)

    // We go do the work in a separate goroutine, the closure has access to the
    // channel we just instanciated so we will be able to manipulate it.
    go func() {
        x, ok := self.soapRequest("GetStatusInfo", statusRequestStringReader(self.upnptype))
        if ok && strings.EqualFold(x.Body.Status.NewConnectionStatus, "Connected") {
            y, ok := self.soapRequest("GetExternalIPAddress", externalIPRequestStringReader(s

            if ok {
                ipString := y.Body.IP.NewExternalIPAddress
                ip := net.ParseIP(ipString)
                if ip != nil {
                    ret <- &ConnectionStatus{true, ip}
                    return
                } else {
                    l4g.Warn("Failed to parse IP string %v", ipString)
                }
            } else {
                l4g.Warn("Failed to get IP address after estabilishing the connection was ok"
            }
        } else if ok && strings.EqualFold(x.Body.Status.NewConnectionStatus, "Disconnected")
            ret <- &ConnectionStatus{false, nil}
        }
        close(ret)
    }()

    // We immediately return the channel to the caller
    return
}

// This method creates a port mapping on the IGD with internal, external ports
// and protocol respectively equal to the passed port argument (bis) and
// protocol
```

```
//
// Errors are indicated by the channel closing before a PortMapping is returned.
// Listeners should therefore check at the very least for nil, better still
// for channel closure.
//
// NOTA BENE the channel closes after a successive PortMapping has been send on
// it, in order to not leak resources.
func (self *IGD) AddLocalPortRedirection(port uint16, proto protocol) (ret chan *PortMapping) {
    ret = make(chan *PortMapping)

    go func() {
        description := fmt.Sprintf("goupnp %s %d %s", self.iface, port, proto)
        _, ok := self.soapRequest("AddPortMapping",
            createPortMappingStringReader(self.upnptype, port,
                proto, self.iface, description))
        if ok {
            portMapping := PortMapping{
                InternalPort: port,
                ExternalPort: port,
                Enabled:      true,
                Description:  description,
                InternalHost: self.iface,
                Protocol:     proto,
            }

            ret <- &portMapping
        }
        close(ret)
    }()

    return
}

// BUG(nhelke): NOT IMPLEMENTED   ALWAYS RETURNS ERROR
//
// Please feel free to submit a pull request to
// https://github.com/nhelke/goupnpc and I will be sure to merge it.
func (self *IGD) DeletePortRedirection(portMappings ...*PortMapping) (ret chan error) {
    ret = make(chan error)
    go func() {
        ret <- errors.New("Sorry, I haven't implemented this yet. " +
            "Feel free to submit a pull request to github.com/nhelke/goupnpc " +
            "and I will be sure to merge it.")
        close(ret)
    }()
    return ret
}

// This method returns a buffered channel which should be iterated over. The
// channel is closed on after the last port mapping, so iterating over the
// channel will not loop forever.
func (self *IGD) ListRedirections() (ret chan *PortMapping) {
    ret = make(chan *PortMapping, 10)

    go func() {
        var (
            ok bool = true
            i  uint = 0
            x  *soapEnvelope
        )
        for ; ; i++ {
            x, ok = self.soapRequest("GetGenericPortMappingEntry",
                portMappingRequestStringReader(self.upnptype, i))
            if ok {
                portMapping := PortMapping{
                    InternalPort: x.Body.PortMapping.InternalPort,
                    ExternalPort: x.Body.PortMapping.ExternalPort,
                    Enabled:      x.Body.PortMapping.Enabled != 0,
                    Description:  x.Body.PortMapping.Description,
                    InternalHost: net.ParseIP(x.Body.PortMapping.InternalClient),
                }
                portMapping.Protocol = ParseProtocol(x.Body.PortMapping.Protocol)
                ret <- &portMapping
            } else {
                close(ret)
                break
            }
        }
    }()

    return
}

// Function for parsing a protocol in string form to protocol type for use with
// this library's methods. Only TCP and UDP are supported.
func ParseProtocol(proto string) (ret protocol) {
    switch {
    case strings.EqualFold("tcp", proto):
        ret = TCP
    case strings.EqualFold("udp", proto):
        ret = UDP
    }
    return
}

// This function returns true if and only if the passed IP address belongs to
// one of the ranges reserved in RFC 1918 for use in private networks
//
```

```go
// This function is only part of this package as it is used internally and is
// public as it is deemed useful for developers to assertain whether or not a
// given external IP address such as one returned by GetConnectionStatus is
// public or not and as creating a standalone package just for this one function
// seemed excessive.
func IsPrivateIPAddress(addr net.IP) bool {
    ip4 := addr.To4()
    if ip4 == nil || !ip4.IsGlobalUnicast() {
        return false
    }
    var (
        aAddr = net.IPv4(10, 0, 0, 0)
        aMask = net.IPv4Mask(255, 0, 0, 0)

        bAddr = net.IPv4(172, 16, 0, 0)
        bMask = net.IPv4Mask(255, 240, 0, 0)

        cAddr = net.IPv4(192, 168, 0, 0)
        cMask = net.IPv4Mask(255, 255, 0, 0)
    )

    return ip4.Mask(aMask).Equal(aAddr) ||
        ip4.Mask(bMask).Equal(bAddr) ||
        ip4.Mask(cMask).Equal(cAddr)
}
```

C.4. github.com/nhelke/goupnpc/goupnp/goupnp_test.go

```go
package goupnp

import (
    "net"
    "testing"
)

func TestIsPrivateIPAddress(t *testing.T) {
    privateAddrs := []net.IP{
        net.IPv4(192, 168, 2, 32),
        net.IPv4(10, 230, 46, 52),
        net.IPv4(172, 22, 8, 61),
    }

    for i := 0; i < len(privateAddrs); i++ {
        if !IsPrivateIPAddress(privateAddrs[i]) {
            t.Errorf("Incorrectly did not identify %v as a private IPv4 Address", privateAddrs[i])
        }
    }

    publicAddrs := []net.IP{
        net.IPv4(184, 85, 61, 15),
        net.IPv4(137, 164, 29, 67),
        net.IPv4(8, 8, 8, 8),
    }

    for i := 0; i < len(publicAddrs); i++ {
        if IsPrivateIPAddress(publicAddrs[i]) {
            t.Errorf("Incorrectly identified %v as a private IPv4 Address", publicAddrs[i])
        }
    }
}
```

C.5. github.com/nhelke/goupnpc/goupnp/backend.go

```
package goupnp

import (
    "bytes"
    "encoding/xml"
    "fmt"
    "io"
    "io/ioutil"
    "net"
    "net/http"

    l4g "code.google.com/p/log4go"
)

type protocol int

func (self protocol) String() string {
    switch self {
    case TCP:
        return "TCP"
    case UDP:
        return "UDP"
    default:
        return "#(Bad Protocol Value)"
    }
}

type deviceElement struct {
    FriendlyName string `xml:"friendlyName"`
    Manufacturer string `xml:"manufacturer"`

    Services []struct {
        ServiceType string `xml:"serviceType"`
        ControlURL  string `xml:"controlURL"`
    } `xml:"serviceList>service"`

    Devices []deviceElement `xml:"deviceList>device",omitempty`
}

type deviceDescription struct {
    XMLName xml.Name `xml:"urn:schemas-upnp-org:device-1-0 root"`

    SpecVersion struct {
        Major int `xml:"major"`
        Minor int `xml:"minor"`
    } `xml:"specVersion"`

    URLBase string

    Device deviceElement `xml:"device"`
}

const (
    connectionTypeStringWANIP  = "urn:schemas-upnp-org:service:WANIPConnection:1"
    connectionTypeStringWANPPP = "urn:schemas-upnp-org:service:WANPPPConnection:1"
)

func statusRequestStringReader(upnptype string) io.Reader {
    return bytes.NewReader([]byte(fmt.Sprintf(statusRequestString, upnptype)))
}

const statusRequestString = `<?xml version="1.0"?>
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/"
s:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<s:Body>
<u:GetStatusInfo xmlns:u="%s">
</u:GetStatusInfo>
</s:Body>
</s:Envelope>
`

const externalIPRequestString = `<?xml version="1.0"?>
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/"
s:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<s:Body>
<u:GetExternalIPAddress xmlns:u="%s"></u:GetExternalIPAddress>
</s:Body>
</s:Envelope>
`

func externalIPRequestStringReader(upnptype string) io.Reader {
    return bytes.NewReader([]byte(fmt.Sprintf(externalIPRequestString, upnptype)))
```

```
}

const portMappingRequestString = `<?xml version="1.0"?>
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/" ` +
    `s:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"><s:Body>` +
    `<u:GetGenericPortMappingEntry xmlns:u="%s"><NewPortMappingIndex>%d` +
    `</NewPortMappingIndex></u:GetGenericPortMappingEntry></s:Body></s:Envelope>
`

func portMappingRequestStringReader(upnptype string, index uint) io.Reader {
    str := fmt.Sprintf(portMappingRequestString, upnptype, index)
    return bytes.NewReader([]byte(str))
}

const createPortMappingString = `<?xml version="1.0"?>
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/" ` +
    `s:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"><s:Body>` +
    `<u:AddPortMapping xmlns:u="%s"><NewRemoteHost></NewRemoteHost>` +
    `<NewExternalPort>%d</NewExternalPort><NewProtocol>%s</NewProtocol>` +
    `<NewInternalPort>%d</NewInternalPort>` +
    `<NewInternalClient>%s</NewInternalClient><NewEnabled>1</NewEnabled>` +
    `<NewPortMappingDescription>%s</NewPortMappingDescription>` +
    `<NewLeaseDuration>0</NewLeaseDuration>` +
    `</u:AddPortMapping></s:Body></s:Envelope>
`

func createPortMappingStringReader(upnptype string, port uint16,
    proto protocol, localAddr net.IP, description string) io.Reader {
    str := fmt.Sprintf(createPortMappingString, upnptype, port, proto, port,
        localAddr, description)
    return bytes.NewReader([]byte(str))
}

type soapEnvelope struct {
    XMLName xml.Name `xml:"http://schemas.xmlsoap.org/soap/envelope/ Envelope"`

    Body struct {
        IP struct {
            XMLName xml.Name `xml:"GetExternalIPAddressResponse"`

            NewExternalIPAddress string
        }
        Status struct {
            XMLName xml.Name `xml:"GetStatusInfoResponse"`

            NewConnectionStatus string
        }
        PortMapping soapPortMapping `xml:"GetGenericPortMappingEntryResponse"`
    }
```

```
}

type soapPortMapping struct {
    Protocol       string `xml:"NewProtocol"`
    ExternalPort   uint16 `xml:"NewExternalPort"`
    InternalPort   uint16 `xml:"NewInternalPort"`
    InternalClient string `xml:"NewInternalClient"`
    Enabled        int    `xml:"NewEnabled"`
    Description    string `xml:"NewPortMappingDescription"`
    Lease          uint   `xml:"NewLeaseDuration"`
}

func (self *IGD) soapRequest(requestType string,
    requestXML io.Reader) (x *soapEnvelope, ok bool) {
    req, err := http.NewRequest("POST", self.controlURL.String(), requestXML)
    if err != nil {
        panic("Programming Error: This hand crafted http.Request should not be bad")
    }
    req.Header.Add("Content-Type", "text/xml")
    req.Header.Add("SOAPAction",
        `"`+self.upnptype+"#"+requestType+`"`)
    req.Header.Add("Connection", "Close")
    req.Header.Add("Cache-Control", "no-cache")
    req.Header.Add("Pragma", "no-cache")

    resp, err := http.DefaultClient.Do(req)
    if err == nil {
        // We got something back, lets not leak it
        defer resp.Body.Close()

        if resp.StatusCode != http.StatusOK {
            return
        }

        body, err := ioutil.ReadAll(resp.Body)
        if err == nil {
            l4g.Debug("SOAP Response:\n%s", string(body))
            err := xml.Unmarshal(body, &x)
            if err == nil {
                ok = true
            } else {
                l4g.Warn(err)
            }
        } else {
            l4g.Warn(err)
        }
    } else {
        l4g.Warn(err)
    }

    return
}
```

C.6. github.com/nhelke/goupnpc/goupnp/backend_test.go

```
package goupnp

import (
    "encoding/xml"
    "testing"
)

const exampleBelkinSOAP = `<?xml version="1.0"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
<m:GetStatusInfoResponse
xmlns:m="urn:schemas-upnp-org:service:WANIPConnection:1">
<NewConnectionStatus>Connected</NewConnectionStatus>
<NewLastConnectionError>ERROR_NONE</NewLastConnectionError>
<NewUptime>194979</NewUptime></m:GetStatusInfoResponse></SOAP-ENV:Body>
</SOAP-ENV:Envelope>
`

func TestSOAPParsing(t *testing.T) {
    var x soapEnvelope
    err := xml.Unmarshal([]byte(exampleBelkinSOAP), &x)
    if err != nil {
        t.Errorf("%v", err)
    } else if status := x.Body.Status.NewConnectionStatus; status != "Connected" {
        t.Errorf("Status incorrectly parsed as %v", status)
    }
}

const exampleBelkinPortMappingResponse = `<?xml version="1.0"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body><m:GetGenericPortMappingEntryResponse
xmlns:m="urn:schemas-upnp-org:service:WANIPConnection:1">
<NewRemoteHost></NewRemoteHost>
<NewExternalPort>5900</NewExternalPort>
<NewProtocol>TCP</NewProtocol>
<NewInternalPort>5901</NewInternalPort>
<NewInternalClient>192.168.2.5</NewInternalClient>
<NewEnabled>1</NewEnabled>
<NewPortMappingDescription>cPM.Port.Map.ee97f96de8c1647a</NewPortMappingDescription>
<NewLeaseDuration>0</NewLeaseDuration>
```

```
</m:GetGenericPortMappingEntryResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
`

func TestPortMappingResponseParsing(t *testing.T) {
    var x soapEnvelope
    err := xml.Unmarshal([]byte(exampleBelkinPortMappingResponse), &x)
    if err != nil {
        t.Errorf("%v", err)
    } else {
        referenceMapping := soapPortMapping{Protocol: "TCP", ExternalPort: 5900,
            InternalPort: 5901, InternalClient: "192.168.2.5", Enabled: 1,
            Description: "cPM.Port.Map.ee97f96de8c1647a"}
        if portMapping := x.Body.PortMapping; portMapping != referenceMapping {
            t.Errorf("Port mapping incorrectly parsed as %#v", portMapping)
        }
    }
}
```

C.7. github.com/nhelke/goupnpc/goupnp/ssdp.go

```
package goupnp

import (
    "bufio"
    "bytes"
    "encoding/xml"
    "errors"
    "fmt"
    "net"
    "net/http"
    "net/url"
    "time"

    l4g "code.google.com/p/log4go"
)

// Returns all local interface IP addresses in the private network range
// They are traversed in the order returned by `net.InterfaceAddrs()`
func localPrivateAddrs() (ret []*net.UDPAddr) {
    addrs, err := net.InterfaceAddrs()
    if err == nil {
        for i := 0; i < len(addrs); i++ {
            if ip, ok := addrs[i].(*net.IPNet); ok {
                if IsPrivateIPAddress(ip.IP) {
                    l4g.Debug("Found private addr %v", ip.IP)
                    ret = append(ret, &net.UDPAddr{ip.IP, 0, ""})
                }
            }
        }
    } else {
        l4g.Warn(err)
    }
    return
}

// This function implements the strict minimum of SSDP in order to discover the
// an IGD on the passed localBindAddr. The function blocks until a UPnP enabled
// IGD is found or timeout of four seconds expires. Timeouts smaller than 3
// seconds are unreasonable This function's behavior is not defined if the
// passed localBindAddr is not an IP address in the private network range. You
// may wish to use goupnp.localPrivateAddrs() to obtain a list of valid such
// addresses for the localhost.
func discoverIGDDescriptionURL(localBindAddr *net.UDPAddr) (u *url.URL, ok bool) {
    const (
        ssdpIPv4Addr = "239.255.255.250"
        ssdpPort     = 1900
        format       = "M-SEARCH * HTTP/1.1\r\n" +
            "HOST: %s:%d\r\n" +
            "ST: %s\r\n" +
            "MAN: \"ssdp:discover\"\r\n" +
            "MX: %d\r\n" +
            "\r\n"
    )

    // These are the various device types we need to M-SEARCH the local subnet
    // for. The last one is a fallback copied from MiniUPnPC's behavior and is
    // unlikely to yield usable results
    //
    // This slice is sorted from most specific device type to the most general.
    // Be advised that the below loop relies on this ordering.
    var deviceTypes = []string{
        "urn:schemas-upnp-org:device:InternetGatewayDevice:1",
        "urn:schemas-upnp-org:service:WANIPConnection:1",
        "urn:schemas-upnp-org:service:WANPPPConnection:1",
        "upnp:rootdevice",
    }

    multicastAddr, err := net.ResolveUDPAddr("udp4", fmt.Sprintf("%s:%d",
        ssdpIPv4Addr, ssdpPort))
    if err != nil {
        panic("Programming error: Our UDPAddr is incorrect")
    }

    conn, err := net.ListenUDP("udp4", localBindAddr)
    var timeout time.Duration = 4 * time.Second
    if err == nil {
        // For each device type, M-SEARCH for it, return the first one found
        // As deviceTypes is sorted from most specific to least specific type
        // returning the first should work fine.
        for i := 0; i < len(deviceTypes); i++ {
            // We write our own request *à la main* as trying to use Go's
            // standard library's HTTP package turns out to be require more
            // code than writing the request by hand, because of the non-
            // standard URL
            requestString := []byte(fmt.Sprintf(format, ssdpIPv4Addr, ssdpPort,
                deviceTypes[i], timeout/time.Second))
            // Allocate a buffer for the response
            buf := make([]byte, 1500)
            // We want to timeout and move on to the next type after a couple of
            // seconds
            conn.SetDeadline(time.Now().Add(timeout))
            // Send multicast request
            conn.WriteToUDP(requestString, multicastAddr)
            // Get a response; the above timeout is still in effect as it
            // should be
            n, addr, err := conn.ReadFromUDP(buf)
            if err == nil {
                // Parse and interpret the response and break if successful
                l4g.Debug("Received %d bytes from %v", n, addr)
                req, err := http.ReadRequest(bufio.NewReader(bytes.NewReader(
                    requestString)))
                if err != nil {
                    // Failure to parse the request represents an assertion
                    // failure as we crafted the request ourselves and have
                    // ensured its validity
                    panic(err)
                }
                resp, err := http.ReadResponse(bufio.NewReader(bytes.NewReader(
                    buf[:n])), req)
                if err == nil {
                    // We got something back, lets not leak it
                    defer resp.Body.Close()
                    l4g.Debug("Discovered device returned:\n%v", resp.Header)
                    // We extract the description URL returned in the Location
                    // header. The UPnP standard ensure
                    urls := resp.Header["Location"]
                    // We must check that the Location header exists as required
                    // by the standard to avoid panicking if we get a bad
                    // response missing a Location header.
                    if len(urls) > 0 {
                        // We have the location, bundle it up into a url.URL
                        // object and return it
                        u, err = url.Parse(urls[0])
                        ok = err == nil
                        return
                    } else {
                        l4g.Warn("Response did not contain Location header:\n%v",
                            resp.Header)
                    }
                } else {
                    l4g.Warn(err)
                }
            } else {
                l4g.Warn(err)
            }
        }
    } else {
        l4g.Warn(err)
    }
    // If we get here we could not find any UPnP devices
    return // ok is false by default, signaling this failure
}

func extractConnectionControlURL(d deviceElement) (upnptype, url string, ok bool) {
    for i := 0; i < len(d.Services); i++ {
        if serviceType := d.Services[i].ServiceType; serviceType == connectionTypeStringWANIP
            serviceType == connectionTypeStringWANPPP {
            return serviceType, d.Services[i].ControlURL, true
        }
    }
    for i := 0; i < len(d.Devices); i++ {
        if upnptype, url, ok = extractConnectionControlURL(d.Devices[i]); ok {
            return
        }
    }
    return
}

func getConnectionControlURL(body []byte) (upnptype, url string, err error) {
    var x deviceDescription
    err = xml.Unmarshal(body, &x)
    if err == nil {
        var ok bool
        upnptype, url, ok = extractConnectionControlURL(x.Device)
        if !ok {
            err = errors.New("Control URL not found")
        } else {
            // The URLs in the DeviceDescription elements are relative
            url = x.URLBase + url
        }
    }
    return
}
```

C.8. github.com/nhelke/goupnpc/goupnp/ssdp_test.go

```
package goupnp

import (
    "testing"
)

func TestDescriptionParsing(t *testing.T) {
    const belkinDescription string = `<?xml version="1.0"?>
<root xmlns="urn:schemas-upnp-org:device-1-0">
    <specVersion>
        <major>1</major>
        <minor>0</minor>
    </specVersion>
    <URLBase>http://192.168.2.1:80</URLBase>
    <device>
        <deviceType>urn:schemas-upnp-org:device:InternetGatewayDevice:1</deviceType>
        <friendlyName>Belkin N150 Wireless Router</friendlyName>
        <manufacturer>Belkin International</manufacturer>
        <manufacturerURL>http://www.Belkin.com</manufacturerURL>
        <modelDescription>Wireless Router with Ethernet Switch</modelDescription>
        <modelName>N150 Wireless Router</modelName>
        <modelNumber>F9K1001</modelNumber>
        <modelURL>http://www.Belkin.com</modelURL>
        <serialNumber>201223GB303099</serialNumber>
        <UDN>uuid:upnp-InternetGatewayDevice-1_0-08863bf24378</UDN>
        <UPC>00000-00001</UPC>
        <serviceList>
            <service>
                <serviceType>urn:schemas-upnp-org:service:Layer3Forwarding:1</serviceType>
                <serviceId>urn:upnp-org:serviceId:L3Forwarding1</serviceId>
                <controlURL>/upnp/service/Layer3Forwarding</controlURL>
                <eventSubURL>/upnp/service/Layer3Forwarding</eventSubURL>
                <SCPDURL>/upnp/service/L3Frwd.xml</SCPDURL>
            </service>
        </serviceList>
        <deviceList>
            <device>
                <deviceType>urn:schemas-upnp-org:device:WANDevice:1</deviceType>
                <friendlyName>Belkin N150 Wireless Router</friendlyName>
                <manufacturer>Belkin International</manufacturer>
                <manufacturerURL>http://www.Belkin.com</manufacturerURL>
                <modelDescription>Wireless Router with Ethernet Switch</modelDescription>
                <modelName>N150 Wireless Router</modelName>
                <modelNumber>F9K1001</modelNumber>
                <modelURL>http://www.Belkin.com</modelURL>
                <serialNumber>201223GB303099</serialNumber>
                <UDN>uuid:upnp-WANDevice-1_0-08863bf24378</UDN>
                <UPC>00000-00001</UPC>
                <serviceList>
                    <service>
                        <serviceType>urn:schemas-upnp-org:service:WANCommonInterfaceConfig:1<
                        <serviceId>urn:upnp-org:serviceId:WANCommonInterfaceConfig</serviceId
                        <controlURL>/upnp/service/WANCommonInterfaceConfig</controlURL>
                        <eventSubURL>/upnp/service/WANCommonInterfaceConfig</eventSubURL>
                        <SCPDURL>/upnp/service/WANCICfg.xml</SCPDURL>
                    </service>
                </serviceList>
                <deviceList>
                    <device>
                        <deviceType>urn:schemas-upnp-org:device:WANConnectionDevice:1</device
                        <friendlyName>Belkin N150 Wireless Router</friendlyName>
                        <manufacturer>Belkin International</manufacturer>
                        <manufacturerURL>http://www.Belkin.com</manufacturerURL>
                        <modelDescription>Wireless Router with Ethernet Switch</modelDescript
                        <modelName>N150 Wireless Router</modelName>
                        <modelNumber>F9K1001</modelNumber>
                        <modelURL>http://www.Belkin.com</modelURL>
                        <serialNumber>201223GB303099</serialNumber>
                        <UDN>uuid:upnp-WANConnectionDevice-1_0-08863bf24378</UDN>
                        <UPC>00000-00001</UPC>
                        <serviceList>
                            <service>
                                <serviceType>urn:schemas-upnp-org:service:WANIPConnection:1</
                                <serviceId>urn:upnp-org:serviceId:WANIPConnection</serviceId>
                                <controlURL>/upnp/service/WANIPConnection</controlURL>
                                <eventSubURL>/upnp/service/WANIPConnection</eventSubURL>
                                <SCPDURL>/upnp/service/WANIPCn.xml</SCPDURL>
                            </service>
                        </serviceList>
                    </device>
                </deviceList>
            </device>
        </deviceList>
        <presentationURL>/index.html</presentationURL>
    </device>
</root>
`

    upnptype, url, err := getConnectionControlURL([]byte(belkinDescription))
    if upnptype != connectionTypeStringWANIP || url != "http://192.168.2.1:80/upnp/service/WA
        t.Errorf("Type: %v, URL: %v, Error: %v", upnptype, url, err)
    }
}
```

C.9. Known issues

1. The GoUPnP package must be compiled with Go version 1.1 or later, because of a bug in the standard "http" package in prior versions of Go.

Appendix D.

Patches submitted to open source projects

During this project, a couple of issues with some of the libraries used came to my attention and I submitted these two patches for them:

D.1. Patch submitted to github.com/nictuku/dht

This patch is discussed in section 4.3

```
From 5eadf6d5a3c57e83e17ce9ceba87b658c46dc553 Mon Sep 17 00:00:00 2001
From: Nicholas Helke <nhelke@gmail.com>
Date: Thu, 30 May 2013 17:56:40 +0200
Subject: [PATCH] Added flag to enable CPU profiling and adapted API to support
 port number 0 (auto-assign)

Profiling
---------
In order to support profiling it was necessary to modify the main loop and
draining function to ensure that the main function exits cleanly.
It turns out the profile is not written to disk if the program does not exit
cleanly.

Auto-assigned port number
-------------------------
I was a little surprised that this was not supported, I guess most torrent
client determine their port number and only then set up the DHT.
Nonethess less I think this is an interesting feature to have.
---
 dht.go                                  | 16 +++++++-
 examples/find_infohash_and_wait/main.go | 66 +++++++++++++++++++++++++++-------
 krpc.go                                 |  2 +-
 3 files changed, 67 insertions(+), 17 deletions(-)

diff --git a/dht.go b/dht.go
index d8c2e8f..6eeddca 100644
--- a/dht.go
+++ b/dht.go
@@ -164,6 +164,13 @@ func (d *DHT) PeersRequest(ih string, announce bool) {
     l4g.Info("DHT: torrent client asking more peers for %x.", ih)
 }
```

```

+// Port returns the port number assigned to the DHT. This is useful when
+// when initialising the DHT with port 0, i.e. automatic port assignment,
+// in order to retrieve the actual port number used.
+func (d *DHT) Port() int {
+    return d.port;
+}
+
 // AddNode informs the DHT of a new node it should add to its routing table.
 // addr is a string containing the target node's "host:port" UDP address.
 func (d *DHT) AddNode(addr string) {
@@ -200,6 +207,11 @@ func (d *DHT) DoDHT() {
         return
     }
     d.conn = socket
+
+    // Update the stored port number in case it was set 0, meaning it was
+    // set automatically by the system
+    d.port = socket.LocalAddr().(*net.UDPAddr).Port
+
     bytesArena := newArena(maxUDPPacketSize, 500)
     go readFromSocket(socket, socketChan, bytesArena)

@@ -228,7 +240,7 @@ func (d *DHT) DoDHT() {
             rateLimit = 10
         }
     }
-    l4g.Info("DHT: Starting DHT node %x.", d.nodeId)
+    l4g.Info("DHT: Starting DHT node %x on port %d.", d.nodeId, d.port)

     for {
         select {
@@ -285,7 +297,7 @@ func (d *DHT) DoDHT() {
                     d.processPacket(p)
                     tokenBucket -= 1
                 } else {
-                    // In the future it might be better to avoid dropping things like ping r
+                    // TODO In the future it might be better to avoid dropping things like p
                     totalDroppedPackets.Add(1)
                 }
             } else {
diff --git a/examples/find_infohash_and_wait/main.go b/examples/find_infohash_and_wait/main.g
index 3582216..16afece 100644
--- a/examples/find_infohash_and_wait/main.go
+++ b/examples/find_infohash_and_wait/main.go
@@ -1,5 +1,5 @@
-// Runs a node on UDP port 11221 that attempts to collect 100 peers for an
-// infohash, then keeps running as a passive DHT node.
+// Runs a node on system selected UDP port that attempts to collect 100 peers for
+// an infohash, then keeps running as a passive DHT node.
 //
 // IMPORTANT: if the UDP port is not reachable from the public internet, you
 // may see very few results.
@@ -17,22 +17,41 @@ import (
     "flag"
     "fmt"
     "os"
+    "os/signal"
+    "runtime/pprof"
     "time"

     l4g "code.google.com/p/log4go"
-    "github.com/nictuku/dht"
+    "github.com/nhelke/dht"
     "net/http"
 )

 const (
     httpPortTCP = 8711
-    dhtPortUDP  = 11221
+    dhtPortUDP  = 0 // 0 to let operating system automatically assign a free port
+)
+
+var (
+    quit       = make(chan bool)
+    interrupt  = make(chan os.Signal)
+    cpuprofile = flag.String("cpuprofile", "", "write CPU profile to file")
 )

 func main() {
     flag.Parse()
+    if *cpuprofile != "" {
+        f, err := os.Create(*cpuprofile)
+        if err != nil {
+            l4g.Critical("Unable to create CPU profile file: %v", err)
+        } else {
+            pprof.StartCPUProfile(f)
+            defer pprof.StopCPUProfile()
+        }
+    }
+
+    flag.Parse()
     // Change to l4g.DEBUG to see *lots* of debugging information.
-    l4g.AddFilter("stdout", l4g.WARNING, l4g.NewConsoleLogWriter())
+    l4g.AddFilter("stdout", l4g.INFO, l4g.NewConsoleLogWriter())
     if len(flag.Args()) != 1 {
         fmt.Fprintf(os.Stderr, "Usage: %v <infohash>\n\n", os.Args[0])
         fmt.Fprintf(os.Stderr, "Example infohash: d1c5676ae7ac98e8b19f63565905105e3c4c37a2\n")
```

```
@@ -62,24 +81,43 @@ func main() {
     go d.DoDHT()
     go drainresults(d)
 
-    for {
-        // Give the DHT some time to "warm-up" its routing table.
-        time.Sleep(5 * time.Second)
+    // Signal handling is necessary so that we exit in a clean state capable
+    // of producing an optional CPU profile
+    signal.Notify(interrupt, os.Interrupt)
 
-        d.PeersRequest(string(ih), false)
+F:
+    for {
+        select {
+        case <-time.After(5 * time.Second):
+            // Give the DHT some time to "warm-up" its routing table.
+            // TODO Possily create a channel to let the DHT adive us when it is
+            // nice and hot
+            d.PeersRequest(string(ih), false)
+        case <-interrupt:
+            break F
+        }
     }
+
+    quit <- true
+    <-quit
 }
 
 // drainresults loops, printing the address of nodes it has found.
 func drainresults(n *dht.DHT) {
-    fmt.Println("========================= DHT")
     l4g.Warn("Note that there are many bad nodes that reply to anything you ask.")
     l4g.Warn("Peers found:")
-    for r := range n.PeersRequestResults {
-        for _, peers := range r {
-            for _, x := range peers {
-                l4g.Warn("%v", dht.DecodePeerAddress(x))
+F:
+    for {
+        select {
+        case r := <-n.PeersRequestResults:
+            for _, peers := range r {
+                for _, x := range peers {
+                    l4g.Warn("%v", dht.DecodePeerAddress(x))
+                }
             }
+        case <-quit:
+            break F
```

```
        }
    }
+    quit <- true
 }
diff --git a/krpc.go b/krpc.go
index e3ba095..6aa1ac6 100644
--- a/krpc.go
+++ b/krpc.go
@@ -196,7 +196,7 @@ func listen(listenPort int) (socket *net.UDPConn, err error) {
     // debug.Printf("DHT: Listening for peers on port: %d\n", listenPort)
     listener, err := net.ListenPacket("udp4", ":"+strconv.Itoa(listenPort))
     if err != nil {
-        // debug.Println("DHT: Listen failed:", err)
+        l4g.Critical("DHT: Listen failed:", err)
     }
     if listener != nil {
         socket = listener.(*net.UDPConn)
--
1.8.3.1

```

D.2. Patch submitted to github.com/monnand/dhkx

This patch is discussed in section 3.3

```
From 0c763f0ff37752f3a48aca7abb363fb9e255388b Mon Sep 17 00:00:00 2001
From: Nicholas Helke <nhelke@gmail.com>
Date: Mon, 1 Jul 2013 14:46:04 -0700
Subject: [PATCH] Added a CreateGroup method to dhgroup.go

Also added an associated test to dhkx_test.go and fleshed out the documentation
in dhgroup.go

Fixes monnand#1
---
 dhgroup.go   | 19 +++++++++++++++++++
 dhkx_test.go | 17 +++++++++++++++--
 2 files changed, 34 insertions(+), 2 deletions(-)

diff --git a/dhgroup.go b/dhgroup.go
index ea3bec4..e4cfd93 100644
--- a/dhgroup.go
+++ b/dhgroup.go
@@ -47,6 +47,10 @@ func (self *DHGroup) GeneratePrivateKey(randReader io.Reader) (key *DHKey, err e
     return
 }

+// This function fetches a DHGroup by its ID as defined in either RFC 2409 or
+// RFC 3526.
```

```
+//
+// If you are unsure what to use use group ID 0 for a sensible default value
 func GetGroup(groupID int) (group *DHGroup, err error) {
     if groupID <= 0 {
         groupID = 14
@@ -77,6 +81,21 @@ func GetGroup(groupID int) (group *DHGroup, err error) {
     return
 }

+// This function enables users to create their own custom DHGroup.
+// Most users will not however want to use this function, and should prefer
+// the use of GetGroup which supplies DHGroups defined in RFCs 2409 and 3526
+//
+// WARNING! You should only use this if you know what you are doing. The
+// behavior of the group returned by this function is not defined if prime is
+// not in fact prime.
+func CreateGroup(prime, generator *big.Int) (group *DHGroup) {
+    group = &DHGroup{
+        g: generator,
+        p: prime,
+    }
+    return
+}
+
 func (self *DHGroup) ComputeKey(pubkey *DHKey, privkey *DHKey) (key *DHKey, err error) {
     if self.p == nil {
         err = errors.New("DH: invalid group")
diff --git a/dhkx_test.go b/dhkx_test.go
index c2a7597..aad31da 100644
--- a/dhkx_test.go
+++ b/dhkx_test.go
@@ -19,13 +19,14 @@ package dhkx

 import (
     "fmt"
+    "math/big"
     "testing"
 )

 type peer struct {
-    priv *DHKey
+    priv  *DHKey
     group *DHGroup
-    pub *DHKey
+    pub   *DHKey
 }

 func newPeer(g *DHGroup) *peer {
@@ -88,3 +89,15 @@ func TestKeyExchange(t *testing.T) {
     }
 }

+func TestCustomGroupKeyExchange(t *testing.T) {
+    p, _ := new(big.Int).SetString("FFFFFFFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD129024E088A67CC74020B
+    g := new(big.Int).SetInt64(2)
+    group := CreateGroup(p, g)
+    p1 := newPeer(group)
+    p2 := newPeer(group)
+
+    err := exchangeKey(p1, p2)
+    if err != nil {
+        t.Errorf("%v", err)
+    }
+}
--
1.8.3.1
```

Appendix E.

GNU General Public License Version 2

As some of the code of this project is licensed under the GNU Public License Version 2 (GPLv2), I am legally required to distribute the license herewith:

rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

under Section 2) in object code or executable form under the terms of
Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable
source code, which must be distributed under the terms of Sections
1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three
years, to give any third party, for a charge no more than your
cost of physically performing source distribution, a complete
machine-readable copy of the corresponding source code, to be
distributed under the terms of Sections 1 and 2 above on a medium
customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer
to distribute corresponding source code. (This alternative is
allowed only for noncommercial distribution and only if you
received the program in object code or executable form with such
an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for
making modifications to it. For an executable work, complete source
code means all the source code for all modules it contains, plus any
associated interface definition files, plus the scripts used to
control compilation and installation of the executable. However, as a
special exception, the source code distributed need not include
anything that is normally distributed (in either source or binary
form) with the major components (compiler, kernel, and so on) of the
operating system on which the executable runs, unless that component
itself accompanies the executable.

If distribution of executable or object code is made by offering
access to copy from a designated place, then offering equivalent
access to copy the source code from the same place counts as
distribution of the source code, even though third parties are not
compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program
except as expressly provided under this License. Any attempt
otherwise to copy, modify, sublicense or distribute the Program is
void, and will automatically terminate your rights under this License.
However, parties who have received copies, or rights, from you under
this License will not have their licenses terminated so long as such
parties remain in full compliance.

5. You are not required to accept this License, since you have not
signed it. However, nothing else grants you permission to modify or
distribute the Program or its derivative works. These actions are
prohibited by law if you do not accept this License. Therefore, by

```
            How to Apply These Terms to Your New Programs

  If you develop a new program, and you want it to be of the greatest
possible use to the public, the best way to achieve this is to make it
free software which everyone can redistribute and change under these terms.

  To do so, attach the following notices to the program.  It is safest
to attach them to the start of each source file to most effectively
convey the exclusion of warranty; and each file should have at least
the "copyright" line and a pointer to where the full notice is found.

    {{description}}
    Copyright (C) {{year}}  {{fullname}}

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License along
    with this program; if not, write to the Free Software Foundation, Inc.,
    51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this
when it starts in an interactive mode:

    Gnomovision version 69, Copyright (C) year name of author
    Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
    This is free software, and you are welcome to redistribute it
    under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate
parts of the General Public License.  Of course, the commands you use may
be called something other than `show w' and `show c'; they could even be
mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your
school, if any, to sign a "copyright disclaimer" for the program, if
necessary.  Here is a sample; alter the names:

  Yoyodyne, Inc., hereby disclaims all copyright interest in the program
  `Gnomovision' (which makes passes at compilers) written by James Hacker.
  {signature of Ty Coon}, 1 April 1989
  Ty Coon, President of Vice

This General Public License does not permit incorporating your program into
proprietary programs.  If your program is a subroutine library, you may
consider it more useful to permit linking proprietary applications with the
library.  If this is what you want to do, use the GNU Lesser General
Public License instead of this License.
```