LAPPEENRANNAN TEKNILLINEN YLIOPISTO
Teknistaloudellinen tiedekunta
Tietotekniikan diplomi-insinöörin koulutusohjelma

Valtteri Kekki

DESIGN AND IMPLEMENTATION OF A PEER-TO-PEER CLIENT FOR DEVICE MANAGEMENT

Työn tarkastajat: Professori Jari Porras, DI Ville Kinnunen

Työn ohjaajat: Professori Jari Porras, DI Ville Kinnunen

ABSTRACT

Lappeenranta University of Technology
Faculty of Technology Management
Master's degree programme in information technology

Valtteri Kekki

Design and implementation of a peer-to-peer client for device management

Master's thesis

2013

105 pages, 9 tables, 22 figures

Examiner 1: Professor Jari Porras

Examiner 2: M.Sc. Ville Kinnunen

Keywords: Device management, peer-to-peer, Skype, hierarchical networks

The aim of this master's thesis was to specify a system requiring minimal configuration and providing maximal connectivity in the vein of Skype but for device management purposes. As peer-to-peer applications are pervasive and especially as Skype is known to provide this functionality, the research was focused on these technologies. The resulting specification was a hybrid of a tiered hierarchical network structure and a Kademlia based DHT. A prototype was produced as a proof-of-concept for the hierarchical topology, demonstrating that the specification was feasible.

TIIVISTELMÄ

Lappeenrannan Teknillinen Yliopisto
Teknistaloudellinen tiedekunta
Tietotekniikan diplomi-insinöörin koulutusohjelma

Valtteri Kekki

Design and implementation of a peer-to-peer client for device management

Diplomityö

2013

105 sivua, 9 taulukkoa, 22 kuvaa

Tarkastaja 1: Professori Jari Porras

Tarkastaja 2: DI Ville Kinnunen

Asiasanat: laitehallinta, vertaisverkot, Skype, hierarkiset verkot

Diplomityön tavoitteena oli määritellä mahdollisimman vähän konfiguraatiota vaativa, mahdollisimman luotettavan yhteyden tarjoava ratkaisu etähallintaohjelmakäyttöön. Vertaisverkkosovellukset ja erityisesti Skype tunnetaan kyvystään toimia lähes missä tahansa verkko-olosuhteissa, joten tutkimuskohde rajattiin näihin teknologioihin. Tutkimustuloksena oli yhdistelmä monitasoista hierarkista verkkoa, jonka rinnalla toimii tietovarastona Kademlia-pohjainen hajautustaulu. Määritelmän lisäksi tuotettiin prototyyppi, jolla osoitettiin määritelmän olevan teknisesti mahdollinen.

Foreword

My heartfelt thanks go out to both Jari Porras and Ville Kinnunen for their invaluable help in the process of writing this thesis and providing the author with physical and mental resources for doing so. Also, enough cannot be said about the courtesy of Miradore Ltd. for providing me the opportunity and trust to work on such an intriguing project.

Observing the bigger picture, there are not many people in my life who would not deserve a credit here. If you feel this paper does justice to both of us, consider yourself included. Yes, this mean YOU! Thank you for being there!

Addendum: nineish months after writing the two previous paragraphs it's only now striking me how arduous a process can be. However, nuttin' to it but to do it. Yeah buddy!

And of course, mom and dad, you're the best. And that means good.

Contents

Symbols and abbreviations

C&C	Command and Control
CAB	Microsoft Compressed Archive Format
CAN	Content Addressable Network
CMDB	Configuration Management Database
CPU	Central Processing Unit
DHT	Distributed Hash Table
GGEP	Gnutella Generic Extension Protocol
GSN	Global Super Node
GUID	Globally Unique Identifier
HTTP(S)	Hypertext Transfer Protocol (Secure)
IIS	Microsoft Internet Information Services
IP	Internet Protocol
IRC	Internet Relay Chat
IT	Information Technology
kbps	Kilobits per second
kBps	Kilobytes per second
LSN	Local Super Node
NAT	Network Address Translation
SCEP	Simple Certificate Enrollment Protocol
SMB	Server Message Block
TCP	Transmission Control Protocol

TLS	Transport Layer Security
TTL	Time To Live
UDP	User Datagram Protocol
UI	User Interface
URL	Uniform Resource Locator

1. Introduction

This section gives a brief introduction to this thesis. First, the background and reasoning as to why this thesis exists in the first place are presented. Then, a high level description of the goals of the thesis work and their scope is discussed and the research questions posed, after which a general description is given of the structure of the rest of this thesis.

1.1. Background

Miradore is a small, fast growing software company with a single product, Miradore Configuration Management, an information technology (IT) asset configuration and life cycle management tool. While heavy investment in research and development has caused the amount of personnel to grow in proportion, the company's strength still lies in its adaptability and rapid, customer oriented development. As the client base is likewise growing and the business environment changing, the company is in a prime position to take advantage of its responsiveness to quickly develop innovative and unique solutions for the device management marketplace.

At any given time there are features which have been requested by customers under development. Recently, new scenarios have been starting to emerge calling for a solution which would need minimal configuration in the style of Skype as well as provide a way for real time communication between helpdesk workers and end users of computers. Additionally, initial plans are being laid for providing the software as a service from a cloud. This thesis discusses creating a new communications architecture for the Miradore client, the device management program running on managed hosts.

As the concept of the Miradore Configuration Management product is moving from a centralized company service to a more service oriented model, the current centralized client-server architecture is becoming dated and rather than changing every tool and feature of the current client to support this change, it was deemed sensible to develop an entirely new communications paradigm both to support current features and to create opportunities for

creating new features. Additionally, adaptation of a peer-to-peer model would allow for much less configuration in software management as installation points, the devices which store installation media, could be distributed instead of running on dedicated hosts, thus providing an opportunity to create automatic load and network traffic balancing.

The main emphases for the new solution are ease of configuration, robust function and scalability of data processing. There should be as little need as possible to build any specialized network infrastructure or to configure special access rules and the data should flow even through network address translators. The client network should itself be able to be used to bear the burden of data gathering and management as opposed to the current model where the central servers create chokepoints which have at times been known to cause congestion and slow user interface (UI) response times when large amount of clients have been simultaneously active.

The internet telephony program Skype is one of the best known peer-to-peer applications and is notorious for its ability to function almost anywhere so long as an internet connection is available with minimal user intervention. Peer-to-peer technology is also pervasive in online content sharing and it is estimated that peer-to-peer applications are the single largest traffic causing category on the internet with the BitTorrent protocol alone causing between 20% and 57% of all traffic (Schulze & Mochalski, 2009). As ease of configuration, data distribution and reliability of communications are the focus points for the new Miradore client solution, the scope of research was chosen to encompass peer-to-peer technologies.

1.2. Goals and scope

The goal of this thesis is to produce an initial architectural specification and a proof-of-concept prototype for a peer-to-peer communication layer for the Miradore client.

The specification consists of the topology of the peer-to-peer network, the communication protocols and associated messages, authentication and encryption schemes both peer-to-peer

and end-to-end, network address translation (NAT) traversal schemes, and descriptions of the operations needed for building the network such as joining, parting, exchanging peers and rebalancing of the topology.

The prototype is a computer program running in the Microsoft Windows environment capable of becoming a node in or the master node of the peer-to-peer swarm. The program will route presence information from all hosts to a master node in the network. For debugging purposes routing information is added to the presence information so the master node always knows the topology of the network. Additionally, the master node can poll hosts to get updated presence information from them. The network shall function as the real client network would so far as featuring joining and parting of hosts from the swarm, detection of unannounced parts and keeping the network balanced and functional despite churn. The prototype does not contain NAT traversal, authentication or encryption capabilities nor necessarily implement the entire specified protocol.

The main research question of this thesis is how to best build a peer-to-peer communication system to solve the challenges in the current architecture of Miradore Configuration Management and support the development of the architecture and the features as described in more detail in section 2.

1.3. Research methods and structure of the thesis

The research done in this thesis has a constructive viewpoint. The goal is to as efficiently as possible find a good solution for the problems presented to end up with an initial specification and a proof-of-concept for an innovative, best-of-breed product.

After this introduction, this thesis is divided into four main sections. The second section presents the current and the vision for the future architecture of Miradore Configuration Management, the product of the company. It also discusses the challenges of the current system and sets goals and requirements for both the design and the prototype for the solution.

In the third section, the discussion focuses on reviewing technologies and products for solving similar problems as the ones presented in this thesis to explore whether they or something like them could be used for these as well. The choices of solutions are then made based on the review.

The fourth section of the thesis consists of the initial specification for the network and the prototype based on the cases presented in the third section.

The fifth section discusses the implementation of the prototype, the problems and the issues that arose during the development and how the practical solutions were formed based on both the specification and the dynamics of the development process.

Finally, the sixth section reviews and discusses the results, asserts whether the original goals were met, how the requirements were satisfied and reviews the whole research and development process. Further development ideas are then presented.

2. The architecture of Miradore Configuration Management

This section discusses both the current architecture of Miradore configuration management, the future vision of a more cloud oriented approach and makes grounds for why this development is happening. The perspective of the discussion is from the point of view of the use cases for both of the architectures to make a case as to why the change is being made and why a peer-to-peer communications model would be preferable for the client program. Current and expected future challenges of the architecture relevant to this thesis are discussed and solutions ideas are presented. Also presented are the additional features and opportunities made possible by this development.

2.1. Current model of operation

This section discusses the current architecture of the Miradore configuration management system. At first, a glance is cast on a high level view of the integrated components after which a closer look is taken on the current client-server polling based communications model. While not using the same communications system, all the features made possible by the current model must also be possible using the new one, leading to requirement S1 as presented in section 2.5.1.

2.1.1. The big picture

The Miradore configuration management system, presented in Figure 1, consists of two primary components: the Configuration Management Database (CMDB) server and a client. The server consists of one or more Microsoft SQL servers and one or more web servers on which Microsoft Internet Information Services (IIS) runs the browser based user interface and a set of data connection interfaces. The client is a custom program running on each managed device, communicating with the CMDB server and performing tasks as described in more detail in section 2.1.2. Additional integrated components are the installation points

which are essentially simply Server Message Block (SMB) shares used to distribute software installation media, one or more of which can be configured for a given subnet.

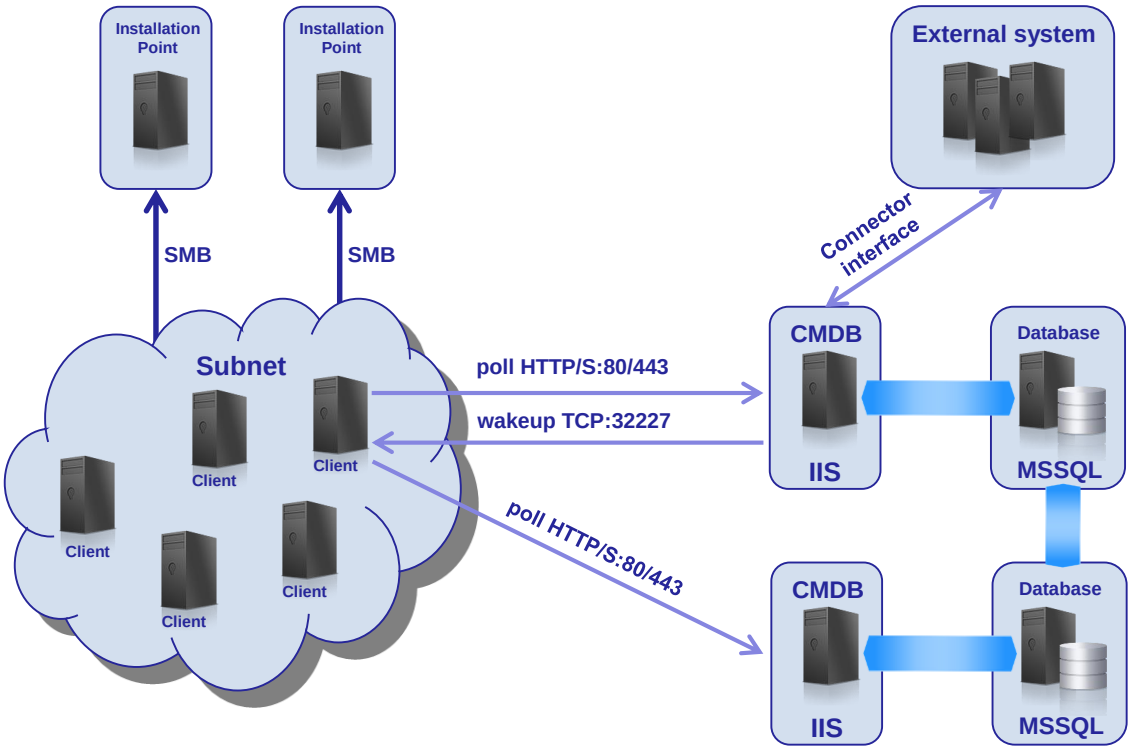

Figure 1 - The high level communications architecture of the Miradore configuration management system

Multiple CMDB servers are configured in a global list and a client will pick one at random when a connection is needed. Installation points can be configured for each subnet and when installing programs, clients will pick one at random. In case no installation points are defined for a subnet, a client will use a global default.

Most data enters the database through a set of interfaces called connectors. The connector interfaces handle a variety of tasks dealing with importing and exporting data from the database such as managing incoming client connections, sending wake-ups to clients, handling the management of a specific platform or importing data from other enterprise systems. All data not manually entered through the user interface will pass through a connector interface before being entered into the database.

2.1.2. Desktop client architecture and client-server communications

Presented in Figure 2, the current desktop client consists of two operationally separate processes running concurrently. These are called the client and the scheduler. The communications model of both of these is based on intermittently polling the server for tasks. The client's responsibilities are concentrated on dynamic tasks such as software installations while the scheduler's sole task is handling the periodical running of external programs and scripts performing such functions as hardware and software inventory gathering. As there is no authentication, the communication model was chosen so there is no way to directly send tasks to the clients. While this leaves open an opportunity for a man-in-the-middle attack, it is still considerably more secure than having the ability to send arbitrary tasks to a program running with administrator privileges without authentication.

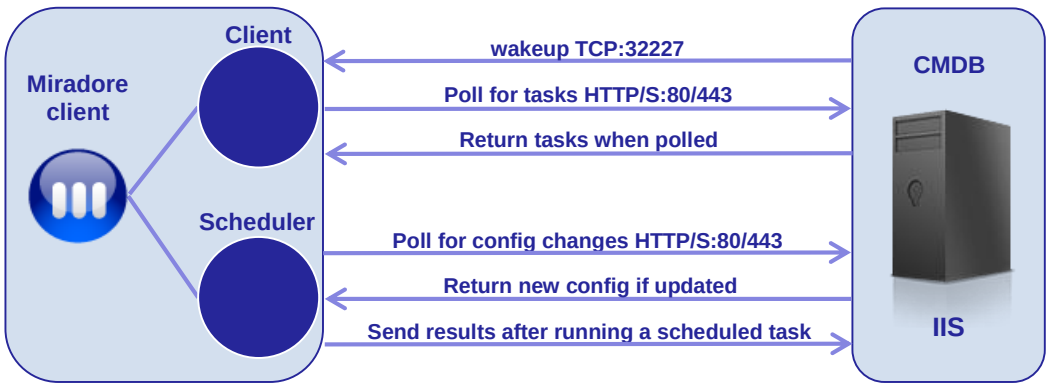

Figure 2 - The components of the client and the communications model

The client periodically polls the server to check whether it has any tasks to run. All communications are initiated by the client to the server. The single exception to this is the server's ability to send up a wakeup message to the client listening on port 32227 but the only function of this message is to get the client to poll immediately. On each poll, the server checks the database for tasks destined for the polling host, which it then sends to the polling client. The client then reports progress and keeps polling until no more tasks are available,

at which point it waits either for a wakeup message or for the polling interval to elapse until the next poll.

Like the client, the scheduler polls the server regularly. However, instead of directly receiving tasks, the scheduler's tasks are defined in an XML configuration file generated at poll time by the server based on the host's designated roles. In the poll message, the scheduler transmits a hash of the current configuration file. The server then constructs a new configuration file, compares its hash with the hash received and sends the polling scheduler a new configuration in case they differ. Based on the configuration file, the scheduler then downloads the programs and scripts it needs to run and as scheduled runs them, sending the results back to the server whenever a task is run.

Currently the operating systems supported by the desktop client are Microsoft Windows, Linux and OS X. Mobile device management is also supported on Windows Phone, Symbian as well as iOS platforms. However, with Symbian fast becoming outdated, iOS and Windows Phone being managed with their own integrated device management solutions and the Android MDM being heavily in development, mobile platforms were scoped out of this thesis. The new system must be implementable in all of the three desktop platforms leading to requirement S2 as presented in section 2.5.1.

2.2. Challenges with the current model

The current model presents some functional problems and limits the use cases to which it is applicable. These challenge cases are based on customer input and reflect the current state of the marketplace.

2.2.1. Network configuration

As previously discussed, the current communications model is based on periodical polls by clients. The polling interval can be configured and is 60 minutes by default. The server does

have the ability to wake up a client to poll immediately by sending a wakeup message via Transmission Control Protocol (TCP) port 32227. If the port on the client is not reachable, the host cannot be contacted and any action performed on the host will remain in queue until the client polls the server the next time.

This presents the problem of port 32227 having to always be open in order to have a reasonable response time for management commands, but the polling brings with it the additional problem of excessive server load with a large number of clients. Each poll generates database traffic and as the number of clients increases, the database can become congested and slow the response time of the entire system, compounding the effect. This issue contributes to requirement S3 of minimal configuration as presented in section 2.5.1.

An emerging network issue is the advent of IPv6 in networks. The original communications code was written for IPv4 networks. While support has recently been added to the client, it is explicitly stated that IPv6 must be supported by the new communications model as stated in requirement S4, presented in section 2.5.1.

2.2.2. Cumbersome communications model

Figure 3 presents an operation flow example of a minimal client task. As can be seen, even for the simplest task, the operation consists of 12 steps, four of which require database I/O and three of which open up a new TCP connection from the client to the CMDB. Additionally, 6kB of communications overhead is caused by each new connection requiring a new HTTPS handshake between the hosts. These scale linearly in proportion to the number of clients, so for example running the presented example tasks for 100 hosts would cause the opening of 300 individual HTTPS connections and 300 database queries.

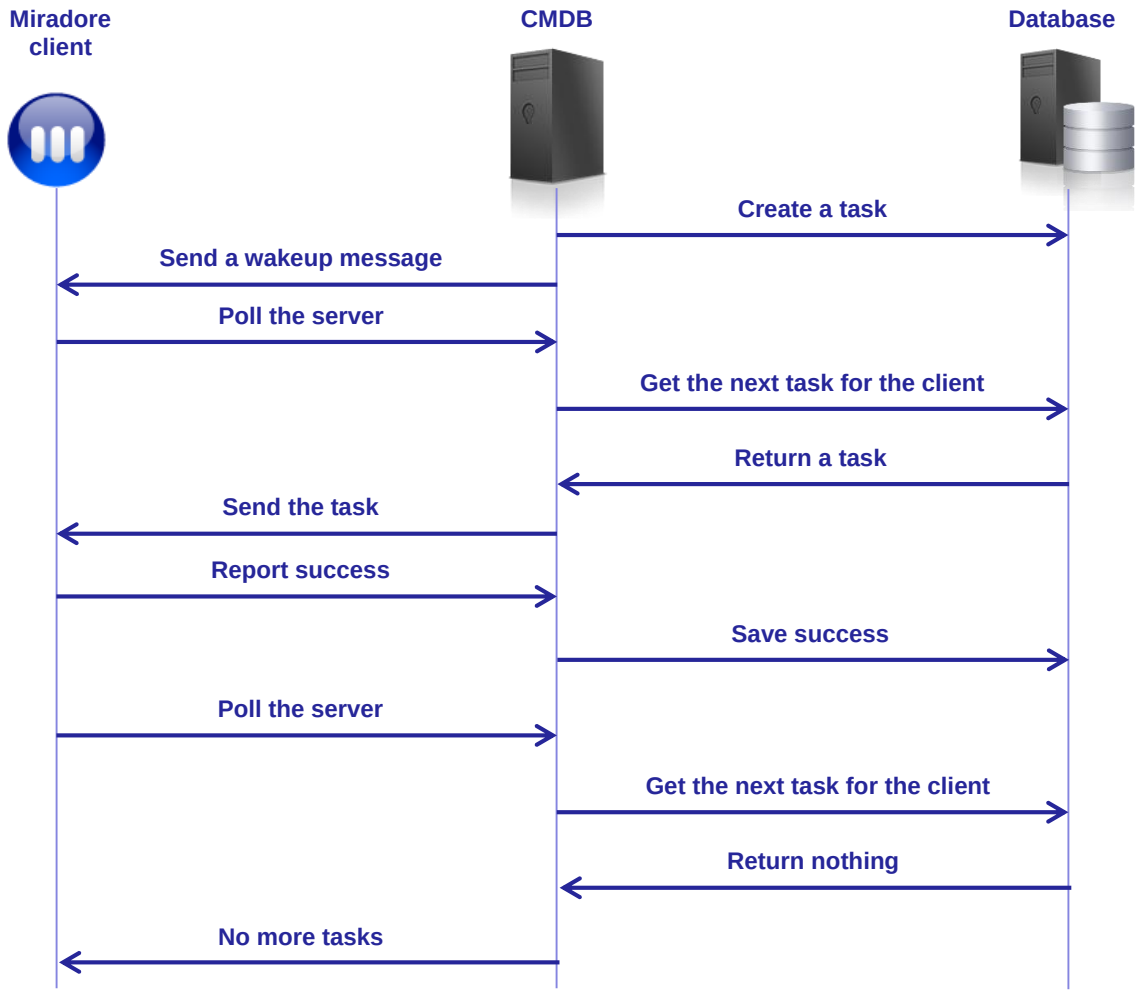

Figure 3 - Operation flow of a client task

The periodical polling of clients alone is a cause of bandwidth and load. Figure 1 presents measurements and calculations of the HTTPS handshakes of hosts polling the CMDB for a single poll for a single host and over 24 hours with the default polling interval of one hour for 1000, 10000 and 30000 hosts. From this can be argued that as the number of clients increases, the traffic generated by just the handshakes can become considerable. Additionally, as each poll causes a database query, additional unnecessary load is produced.

Table 1 - Traffic generated by client polls

Scenario	HTTP	HTTPS	HTTP + Proxy	HTTPS + Proxy
single host, single poll	3 411 bytes	6 378 bytes	7 596 bytes	9 178 bytes
1000 hosts, 24 polls	78 MB	146 MB	174 MB	210 MB
10000 hosts, 24 polls	781 MB	1460 MB	1739 MB	2101 MB
30000 hosts, 24 polls	2.3 GB	4.3 GB	5.1 GB	6.2 GB

While the scheduler's polling also causes similar overhead, it has not been measured. A larger problem caused by the scheduler arises from the timed nature of the tasks combined with the rhythm of the human calendar. Table 2 presents the average amount of data produced by some inventory scans, measured from 12 laptop and desktop computers in the Miradore internal network. The chosen scans each have a running interval of 12 hours apart from the file scan which is run once a week. As computers are typically turned off for the night and back on in the morning, this can cause a large number of hosts to post their inventories in a very short time span. While the measurements show the data being around 2MB per host, assuming the measurements represent a global average of a real world deployment, would mean around 20GB of data for 10000 hosts in the worst case. As this data is not simply stored but parsed and entered into database, the load caused by this can be tremendous and while no measurements have been performed in customer deployments, has been known to cause large issues with system responsiveness. This contributes to requirements S5 and S6 of as presented in section 2.5.1.

Table 2 - Average amount of data produced by inventory scans

Inventory name	amount of data
Add/Remove programs	186 475 bytes
Hardware	23 554 bytes
Plug and Play devices	82 605 bytes
Files	1 679 297 bytes
All combined	1 971 931 bytes

If the communications model could be changed from the current one where the client or the scheduler is always the one to initiate connection to one where the server could directly contact clients and send tasks to them, the operational complexity could greatly be decreased as illustrated in Figure 4. Also, if the polling of clients and schedulers would to become obsolete, bandwidth and server resources could be conserved. Furthermore, if the scheduling of tasks could be done in the server end instead of timing the tasks on clients or if the data could be buffered on clients to be requested on demand, the responsiveness issue caused by the scheduled inventory runs could be alleviated. While traditional load balancing solutions exist and Miradore supports multiple front-end servers, a solution which would scale itself would be much preferable, contributing to specification requirements S3 and S7 as presented in section 2.5.1.

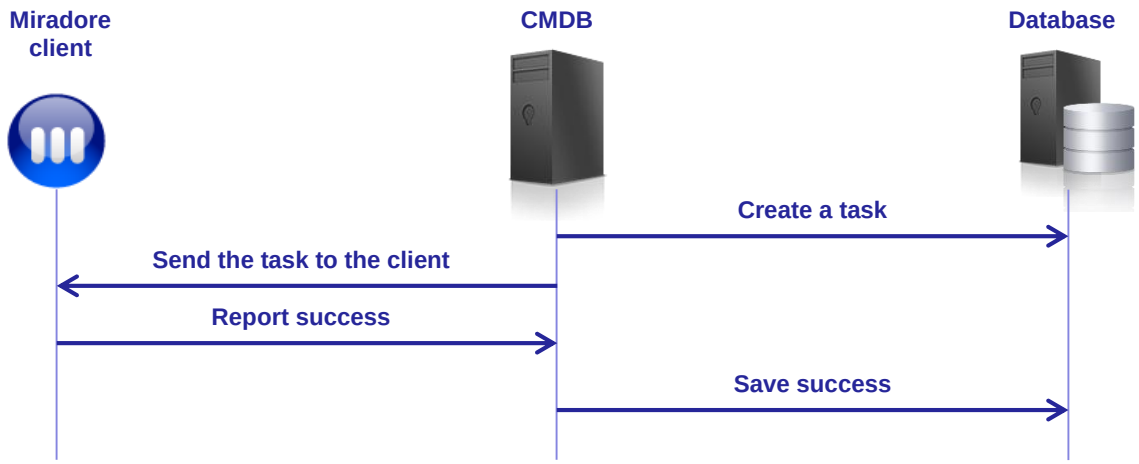

Figure 4 - A more simple model of communication

2.2.3. Diaspora of devices

As organizations are becoming more and more distributed and mobile, the building of fixed infrastructure is getting infeasible or starting to cause intolerable overhead. While a centralized solution residing in the intranet was adequate for an environment of stationary desktop computers, it is becoming more and more cumbersome with short lived ad-hoc organizations. Device management, however, is not something corporations wish to

abandon. Because of this development, there is great demand for a solution for managing a swarm of devices connecting through whatever internet link they possess without building any specialized infrastructure for the purpose.

The challenge doing this with the current system is that all devices need not only to have access to the CMDB server but for the management features to work properly, also be available for contacting by it. As this has proven hard to achieve even inside internal corporate networks, it has been discovered next to impossible to achieve when dealing with devices in random networks. While setting up a VPN is a possibility, not everyone will keep their device connected to the VPN at all times nor will all devices be wanted inside networks. While Miradore clients do support functioning through a proxy or a gateway, this still brings with it the necessity to configure one, the avoidance of which could be a large selling point for the product. The diaspora of devices contributes to requirements S7, S8 and S9 and must be taken into account also with S3 as presented in section 2.5.1.

2.2.4. Lack of a real time connection

Currently, there is no real time connection between the Miradore server and clients. As previously described, the clients and schedulers poll the server periodically to update their online status and ask whether they have any jobs pending. Thus, even if a client appears to be online in the UI, there are no guarantees it is available as the online status currently only signifies that the client has polled the server sometime within a set period.

While annoying, real time online status information is a minor feature. However, a much requested feature which cannot be implemented due to this issue is instant messaging. Both helpdesk workers and administrators would much appreciate the possibility to chat with a user at a given host both for assistance and information purposes, up to being able to use a screen sharing application through the user interface to remotely access a managed host. The new communications model should be such that these are made possible as stated in requirement S10 presented in section 2.5.1. As a real time connection would also increase the reliability of connection to devices moving through networks in a rapid pace, this also contributes to requirement S9.

2.2.5. Lack of bandwidth management

The current architecture has no support for bandwidth management. The clients themselves do not limit bandwidth usage in any way neither for themselves nor for the entire swarm. Additionally, installation points do not currently support any bandwidth limitations so even if an installation point has the needed processing power to serve all the clients needing media, it may end up congesting the network instead. While this can be circumvented by taking into account the amount of resources a given installation point has, it is very labor intensive for an administrator to keep track of installations and incrementally send more instead of just spending a single click to distribute a software package to every recipient at once.

The challenge is to design the new architecture in a way so it is possible to gather bandwidth information so it is possible to easily use it to create a bandwidth management solution. In the best case, this solution would with total automation optimize the bandwidth usage so the network remains responsive for all hosts. While the building and specification of this application is outside of the scope of this thesis, the need for it will have to be taken into account in the design as stated in requirement S11 presented in section 2.5.1.

2.2.6. Encryption and authentication

Presently, HTTPS is the only encryption scheme supported in the client-server communications. To prevent eavesdropping, a secure encryption scheme must be supported also in the new communications model as stated in requirement S12 in section 2.5.1.

The authentication is currently somewhat lacking. However, as all of the clients connect to the same central server, there is no way for any of them to perform a man-in-the middle attack. However, when routing administrative commands through a network of nodes, authentication is required to prevent this. With authentication, all tasks could be verified to

be genuine and thus better security would be achieved. The requirement for authentication is an important one, presented as S13 in section 2.5.1.

2.3. Cloud vision and the peer-to-peer client solution

While the new client architecture itself is a large change, it is only a part of a larger architectural development in the works. The vision of Miradore for the future of device management is that of ever increasing diversity and mobility. The variety will increase in devices and platforms as well as both physical and hierarchical structures of corporations. Organizational trends are pointing towards more and more ad-hoc and dynamic environment with an ever increasing variety of devices located all over the world.

In this envisioned dynamic environment where an internet connection is the only constant, Miradore will provide a service for managing devices throughout their life cycle with emphasis on simple adoption, ease of use, reliability and powerful features based on customer needs.

This section discusses on a high level the conceptual visions the new architecture will have to fulfill, with measurement numbers provided where available and applicable. The discussion is based on the challenges presented in section 2.2. with additional exploration of exploitable opportunities.

2.3.1. Simple adoption and ease of use

Simple adoption is the key factor every component of the new architecture needs to support. If a new user wants to start using Miradore, all they should have to do is register with their information to get access to a Miradore CMDB instance and start distributing client download links to users for this reason the requirement S7 is assigned the highest priority.

For both the client and the entire system the central point of this requirement is minimal requirement for configuration. Currently the typical setup time for a Miradore server is one workday assuming all network preparations have already been done and the persons setting the system up know what they're doing. In the real world, all the preliminary setup steps such as configuring the network and applying for the necessary certificates can take upwards of weeks in a large organization. As opposed to this, the new architecture should enable Miradore device management to be ready for use immediately, and solving the challenges presented in section 2.2.1. would greatly aid in it.

While this movement towards a service based model will likely entail also making large changes to the server end setup process, those are out of the scope of this thesis. Thus, the server service setup is simply assumed to have become a possibility in the context.

2.3.2. Reliability

Miradore device management must be able to be relied upon. With increasing mobility of devices the number of lost devices both due to malicious third parties and simple negligence is also on the increase and lost devices will have to be locked or erased to prevent unauthorized access to data and systems. However, as the devices and people are becoming increasingly mobile and the popularity of the bring-your-own-device (BYOD) scenario is on the rise, users may not be willing to be constantly connected to the corporate VPN. Also, as is especially often the case with user-owned devices, not all of them are wanted inside the corporate network. Furthermore, for some organizations VPN configuration is seen as an insurmountable overhead to be avoided.

This creates a demand for a communication scheme which requires no logging into corporate systems while still retaining as high a reliability as possible and enables two-way communications for the purpose of solving the challenges presented in sections 2.2.1; 2.2.2. and 2.2.3 and contributing to requirements S6, S8, S9 and S16.

2.3.3. Management features

As currently is the case, Miradore must in the future also be able to provide industry standard management features such as asset management, software and hardware inventory gathering, usage statistics gathering and software distribution. A peer-to-peer based communications model would create opportunities for advantages to all of these.

One of the most basic pieces of information known about an asset is its online status. While gathered also in the current system, it only signifies the device has been seen polling within a given time period. If the architecture was changed so the devices wouldn't poll the server but the peer-to-peer swarm could be accessed at any given time, the online status information could be updated in real time.

Another opportunity having the ability to access any up host at any given time would be the ability to store much more data about them.

Table 3 presents a scenario in which the histories for central processing unit (CPU) load, network activity, memory usage and disk usage are kept over time averaging one record a minute for each. The scenario assumes a single core CPU and an optimal way to store the data such that overhead from storage can be ignored.

Table 3 - Asset data measurement scenario

measurement	records/minute	bytes/sample	data/day	records/day
CPU load	1	4	5 760 bytes	1440
Network activity	1	4	5 760 bytes	1440
Memory usage	1	8	11 520 bytes	1440
Disk usage	1	8	11 520 bytes	1440

Table 4 presents further calculations on the amount of data and amount of records needed to store the data presented in the scenario of

Table 3 for 1, 1000, 10000 and 30000 hosts over time intervals up to a year. As the number of hosts increases, both the amount of data and the number of records quickly grow very large. While centrally storing this data is possible, at least the number of records would likely make keeping the data up to date in real time infeasible. This contributes to requirement S15 as presented in section 2.5.1.

Table 4 - Amounts of data and records for asset data measurements over time for amounts of hosts

hosts	data/day	data/month	data/year	records/day	records/month	records/year
1	34 kB	1013 kB	12 MB	5760	172800	2102400
1000	33 MB	989 MB	12 GB	5760000	172800000	2102400000
10000	330 MB	10 GB	117 GB	57600000	1728000000	21024000000
30000	989 MB	29 GB	352 GB	172800000	5184000000	63072000000

Furthermore, if as much as possible of the asset data currently stored in the database could be stored on the assets themselves, database load could be reduced. In currently deployed systems, database load is the single largest cause of poor responsiveness.

Software distribution is also a scenario which can greatly benefit from a peer-to-peer communication model. The current media management scheme requires configuring dedicated installation points for media distribution and configuring packages to use some installation point for media delivery. A peer-to-peer model would allow for the distribution of installation points so hosts could share the burden of media distribution.

Table 5 presents a scenario in which a 10MB update is pushed for 1, 1000, 10000 and 30000 hosts from a single installation point. Overhead data means the overhead caused by the client first being woken up and polling the CMDB and then keeping it updated of the installation status. Payload data means the amount of data the installation point needs to send. As can be

seen, the amount of data grows in linear proportion to the amount of hosts. This is known to sometimes cause an installation point to either crash trying to serve too many concurrent clients or to slow down. Also, if the installation point and the network can handle the load, a large number of clients reporting on their installation progress concurrently may cause large CMDB load as a lot of information has to move in and out of the database. While both of these problems can be avoided by manually configuring multiple installation points and installing the package to a limited number of hosts at the same time, if it was possible to build a self-balancing peer-to-peer system to distribute the content between the peers and not just directly from the installation point as well as buffer the reporting data so all the hosts wouldn't have to concurrently poll the CMDB, both of these problems could be scalably alleviated without requiring manual effort each time.

Table 5 - Amount of data an installation point has to send distributing a 10MB package to a number of hosts

hosts	overhead data	payload data
1	20 kB	10 MB
1000	20 MB	10 GB
10000	195 MB	98 GB
30000	586 MB	293 GB

2.3.4. Opportunities for new features

The new communication model presents some possibilities for entirely new features. There have been requests to make it possible for helpdesk workers to chat with users via Miradore. As the new architecture would allow for real time communication, this would also be possible to implement.

Currently, each Windows based Miradore client posts its inventory data at set, configurable intervals. These inventories are Add/Remove programs inventory, Plug and Play device inventory, file inventory and hardware inventory. Table 6 presents a comparison of all four inventories from 12 windows laptop and desktop hosts selected from the internal network of Miradore packed individually and together in the Microsoft Windows compressed archive

format (CAB). From this comparison it can be seen that while some inventories benefit less from getting packed together, over three-fold improvement in the amount of data transferred might be achieved by packing all inventories of multiple hosts together before sending them off to the CMDB.

Table 6 - Windows client inventory data amounts, packed into individual archives vs. packed into a single archive

Inventory name	amount of data, individually packed	amount of data, packed together	benefit percentage
Add/Remove programs	186 475 bytes	179 787 bytes	104 %
Files	1 679 297 bytes	278 412 bytes	603 %
Hardware	23 554 bytes	8 146 bytes	289 %
Plug and Play devices	82 605 bytes	73 750 bytes	112 %
All combined	1 971 931 bytes	540 095 bytes	365 %

2.4. Peer-to-peer vision

The challenges presented in section 2.3. demand a solution which seem to combine aspects of distributed databases, file sharing and instant messaging. As many peer-to-peer technologies are used in similar situations, such as Skype for instant messaging and BitTorrent for file sharing, this thesis explores these technologies as the solution candidate and aims to create both a prototype and a specification for a complete product.

The vision of the new communications system is very much inspired by Skype: what is wanted is a network that just works. With minimal configuration and simple installation, a client will begin to function, offering the current features as well as additional opportunities and performance benefits.

2.5. Requirements

Based on the problems presented in section 2.2. and the vision presented in section 2.3; the following preliminary requirements were drafted. The requirements are split into two categories, one for the architecture and another for the prototype. Additionally, the constraints of the prototypes are presented. As all of the requirements are important, the priority value denotes rather the priority in the order of implementation, with high priority implying the concept should be proven with the prototype. The highest priorities are the reliability of the system, that is, keeping the clients connected to the network as much of the time as possible and that this should require as little configuration as possible. These two requirements, in addition to their high priority, are marked with an asterisk.

2.5.1. Requirements for the specification

Number	Priority	Requirement	more information
S1	medium	The design must be able to support all existing functionality.	2.1.
S2	low	The system must be implementable on Windows, Linux and OS X.	2.1.2.
S3	high	The network must be scalable up to tens of thousands of hosts in the network without causing excessive load on any host or any point of the network.	2.2.1; 2.2.2; 2.2.3; 2.2.5; 2.3.3.
S4	medium	IPv6 must be supported.	2.2.1.
S5	medium	The system must support distribution of arbitrary data across the network with all clients being able to query and access that data.	2.2.2; 2.2.5; 2.3.3;
S6	high	The network must be able to cope with the circadian rhythm of much of the nodes, remaining reliable even during times of high churn.	2.2.2; 2.3.2.
S7	high*	The system should require as little configuration as possible, preferably none.	2.2.2; 2.2.3; 2.2.5; 2.3.1.

S8	medium	Communications must function even through NATs.	2.2.3; 2.3.2.
S9	high*	The system must be able to keep clients connected to the management network even during rapid movement of hosts between physical networks.	2.2.3; 2.2.4; 2.3.2.
S10	medium	The design must be able to support instant messaging from the central user interface to any host connected to the network.	2.2.4.
S11	medium	There must be a possibility to build support for bandwidth management so even when a large number of clients access a large chunk of data, such as an installation media, neither any parts of the network nor any hosts become congested.	2.2.5.
S12	low	The network must be able to support encrypted communications between nodes.	2.2.6.
S13	low	The network must be able to support host authentication.	2.2.6.
S14	low	The network must be able to support hosts authenticating the network for genuinity, i.e. that it really is the network they wish to join.	2.2.6.
S15	low	The system must be able to support efficient storing of detailed host information histories on the hosts themselves and reporting this data to the master node when queried. Such data may include but is not limited to CPU usage, memory usage and network utilization.	2.3.3.
S16	medium	In case an operation fails, an error message must be generated.	

2.5.2. Requirements and constraints for the prototype

The requirements and constraints for the prototype were scoped based on the previously defined requirements for the specification and their priorities. As stated, its purposes are to serve as a proof of concept as well as serve as a part of exploratory product development. As the nature of the development is exploratory, the requirements and constraints are not assigned priorities and are defined in a loose fashion.

Number	Requirement
PR1	The purpose of the prototype is to function as a proof of concept for the network topology. The topology implemented by the prototype must match that defined in the specification for the final product.
PR2	It must implement a host joining from the network
PR3	It must implement a host parting from the network, both announced and unannounced.
PR4	For evaluation purposes it must implement gathering of presence and routing information. Each host sends its presence information to the designated master node and each host participating in the routing adds its identity information to the message. The master node can then display the presence information of hosts and the topology of the network.

Number	Constraint
PC1	The entire specified protocol will not be implemented.
PC2	The prototype needs function only in Microsoft Windows.
PC3	The prototype needs not implement NAT traversal.
PC4	The prototype needs not support IPv6.

3. Technology review

This section discusses the possibilities of solving the problem based on the latest science as well as existing products. For each studied technology a summary is given and the functionality, such as routing and the joining and parting of nodes is described. Also, performance numbers are briefly presented where available.

After the presentation of the individual technologies, the discussion section asserts the options and presents the arguments for the choice of technologies to be used in the prototype, the specification as presented in section 4 and the final solution.

The reviewed technologies include theoretical solutions to specific problems, implementations of complete peer-to-peer systems, file sharing solutions such as BitTorrent as well as instant messaging systems such as Skype. Due to their similar nature to the device management scenario, botnets are also briefly explored.

3.1. Distributed hash tables

A distributed hash table (DHT) is eponymously a hash table, mapping keys to values, but with the key-value pairs stored on multiple hosts on a network. In a peer-to-peer network, a DHT can be used for example to solve the problem of locating the node or nodes responsible for a particular piece of data by mapping the list of their IP addresses to a hash of that data. The challenge in the this context is that many networks spend much of their time in a dynamic state with hosts joining the network and failing or for other reasons leaving as they please. Regarding this thesis, the solving of this problem is particularly relevant for the requirement of distributing arbitrary data on the network for every member to access.

A distributed hash table is a central feature in many of the solutions presented in this section, including the Content Addressable Network (CAN) (Ratnasamy, et al., 2001), Chord (Stoica, et al., 2001), Pastry (Rowstron & Druschel, 2001), Tapestry (Zhao, et al., 2001) and

Kademlia (Maymounkov & Mazières, 2002), and while not originally a part of the specification, has been implemented also by many BitTorrent clients as well as adopted as a part of the official BitTorrent protocol to reduce the reliance on a central tracker, allowing for decentralized searching of peers.

3.1.1. Content Addressable Network

CAN (Ratnasamy, et al., 2001) is a distributed system that maps keys to values in a hash table like manner, effectively implementing a DHT. It boasts scalability, robustness and the ability to form networks of low latency. While the original definition of a CAN is "a scalable indexing mechanism" (Ratnasamy, et al., 2001) and thus all distributed hash tables could be interpreted as such, this section discusses the original implementation.

CAN was conceived at a time when peer-to-peer file sharing was an emerging technology with Napster and Gnutella being popular networks. These two early networks had problems with scalability, with Napster being dependent on the central servers for content searching and Gnutella using a network wide flood for searching. CAN tries to solve these problems by providing a true peer-to-peer system without a central server which has a scalable method for indexing files. This indexing system is called a Content Addressable Network. (Ratnasamy, et al., 2001)

The topology of a CAN network is a virtual Cartesian coordinate system on a d-dimensional torus, a wrapping system represented by a circle in the 1-dimensional case and as presented in Figure 5, a doughnut shape in the 2-dimensional case with higher dimensions possible as well. The space on this torus is divided amongst the nodes with each node being responsible for an area called a zone, all zones combined form the entire surface and none overlap. Nodes are defined as neighbors if their coordinate spaces overlap in d-1 dimensions and are adjacent to each other in one. In the 2-dimensional case neighboring nodes would be represented by two rectangles sharing an edge for example in Figure 5, node 4 would be neighbors with nodes 1, 2 and 5. (Ratnasamy, et al., 2001)

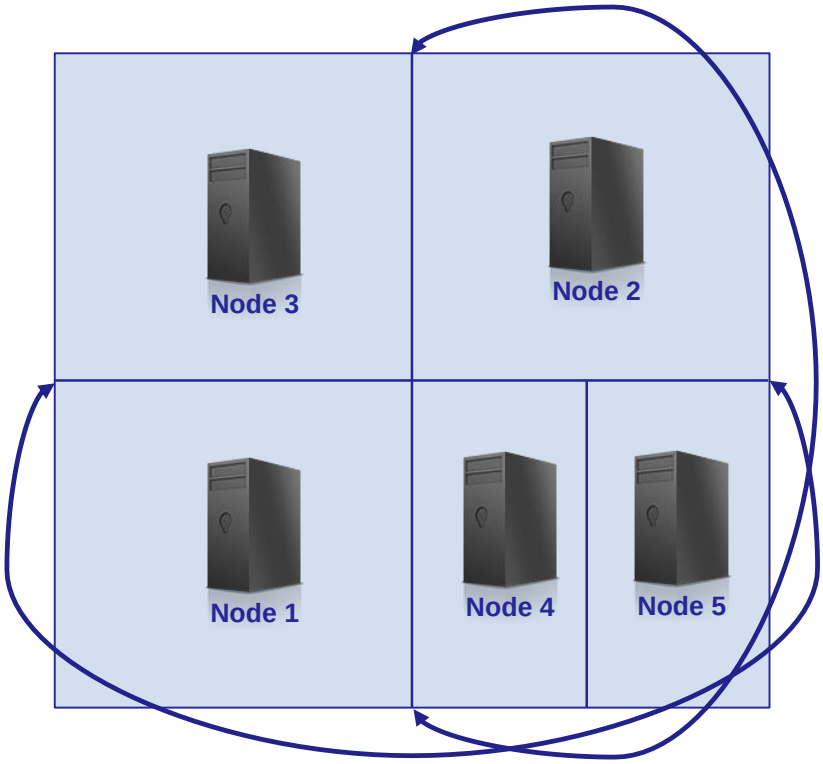

Figure 5 - CAN topology

The key-value pairs are stored in this space by mapping the keys onto a coordinate point in the torus via a uniform hash function and storing them on the node which owns that point of the coordinate space. For additional reliability, each node may exist in several independent coordinate systems so the key-value pairs are likely to be replicated on different hosts in different systems. (Ratnasamy, et al., 2001)

Each node keeps a record of the contact information of its neighbors and if an encountered message is destined to one of these hosts, routes it to them. Otherwise messages are routed in a greedy manner to the neighbor whose coordinates are closest to the target coordinates. If the next hop in the route is unreachable, the message is routed to the next closest neighbor. If all the neighbors of a host closer to the target are for some reason down, a node can use an expanding ring search to look for nearby nodes which are closer to the target and from which greedy forwarding can again resume. The routing performance depends on the dimensionality of the coordinate space, the number of nodes in the network and the current

partitioning conditions. For a *d*-dimensional, equally partitioned space with *n* nodes, the average number of hops is $\left(\frac{d}{4}\right) * \left(n^{\frac{1}{d}}\right)$ with a routing table size of $2d$. (Ratnasamy, et al., 2001)

To join a CAN mesh, a host must be able to find a node already in the network, find a node from the network whose zone will be split and then notify the neighbors of the new zone so the routing information is updated to include it. CAN itself does not specify how a host wishing to join the network should discover nodes already in it, but it is suggested a group of bootstrap nodes with lists of nodes may be used for such discovery purposes. (Ratnasamy, et al., 2001)

After finding a node in the network, the new host randomly chooses a point in the CAN coordinate space and sends a join request to that point. The hosts in the CAN then forward the message via the greedy routing mechanism until it reaches the host responsible for that point. The host responsible for the zone then splits it in half along a dimension and assigns one half to itself and one to the host joining the network, also transferring the key-value pairs residing in the new node's zone to it as well as the IP addresses of the zone's neighbors. The dimension along which the splitting is done follows some ordering so when a node leaves the network the process can be reversed. After assigning the new node its zone, the node whose zone was split updates its own neighbor information to correspond to the new state of the network. (Ratnasamy, et al., 2001)

After the zone has been successfully split and transferred from the control of one node to two zones controlled by two nodes, the neighbors of the new zones must be notified so their routing information becomes correct. This information is also periodically refreshed.. As the number of neighbors a host has depends on the number of dimensions *d* in the CAN coordinate space, it is never needed to update the information for more than *O(d)* hosts regardless of the number of hosts in the network. (Ratnasamy, et al., 2001)

A node departing the network normally relinquishes its key-value pairs and its zone and hands them over to one of its neighbors so the neighbors' zone and the departing node's zone

can be merged together into a valid new zone. If this is not possible, the zone and the key-value pairs are passed to the neighbor whose zone is currently the smallest and that node will handle both of the zones. (Ratnasamy, et al., 2001)

Should a node fail unexpectedly, it is detected by its neighbors by the lack of update messages. As a node detects one of its neighbors to be down, it activates a countdown timer which is the shorter the smaller the node's zone is. After the countdown, the node sends a takeover message to the failed node's neighbors ending their countdown timers and effectively taking over the zone. In case of multiple adjacent nodes failing, it is possible that the takeover mechanism causes the CAN to become inconsistent. In such cases, before initiating the takeover, a host must perform an expanding ring search to determine how large the failed area is and where its neighbors lie. (Ratnasamy, et al., 2001)

These operations sometimes lead to a single node being responsible for more than one zone. To prevent fragmentation, it is preferable that one node would only ever be responsible for one zone. For this, a background process for zone reassignment is proposed. (Ratnasamy, et al., 2001)

While a sound solution on paper, no publicly available implementations of CAN seem to exist, and furthermore no real world applications seem to be using it. Thus, a shadow is cast on the credibility of the technology as the implementation of choice.

3.1.2. Chord

Chord (Stoica, et al., 2001) is another peer-to-peer scheme the main function of which is the storage of key-value pairs, or implementing a DHT. It is specified to scale well even to very large network sizes. Joining and both announced and unannounced parting of nodes are quickly processed to maintain efficient and reliable operations and availability is maintained even during times of high churn, while adding some lookup latency due to partially out of date routing information.

Chord uses a hash function, such as SHA-1 (U.S. Department of Commerce, 1995), to assign all nodes and keys an *m* bits long identifier. The identifier of a node is its hashed IP address and the identifier of a key its hash, thus making the key and node identifier spaces identical. The identifier length and the hash functions need to be chosen so the probability of collision is very low for both keys and nodes. A key is assigned to the node whose identifier is equal or follows next to it on the identifier line which wraps around itself forming a circle called the Chord ring as illustrated in Figure 6. When a new node joins the network, some of the keys previously assigned to its successor in the ring now become assigned to the new node. When a node leaves, its keys are assigned back to its successor. (Stoica, et al., 2001)

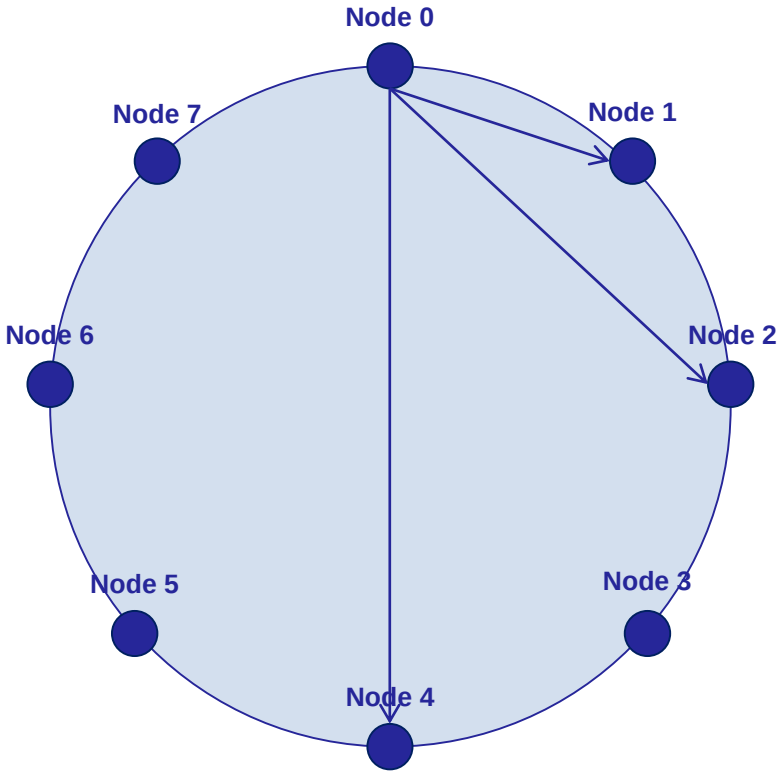

Figure 6 - a chord ring and the finger table of node 0

While routing in the Chord ring is possible with each node knowing their successor this would yield poor performance as in the worst case data would have to be routed through all nodes in the ring before reaching its destination. Because of this, nodes in the Chord ring

keep additional routing information in an additional finger table. The entries in this table, called fingers, spread to a number of hosts with the maximum being the number of bits in the identifier. For an m bit identifier space, the ith entry in a node's finger table contains information of the first node that succeeds the node by at least 2^{i-1} in the identifier space and $1 \leq i \leq m$, as illustrated in Figure 6. For example node 0's finger table in a saturated 8 bit identifier space would contain fingers to nodes 1, 2, 4, 8, 16, 32, 64 and 128. As can be seen, nodes need to store a relatively small number of information, only 160 even for a fully saturated SHA-1 based network. Messages are forwarded by nodes to either the target node or to the node in the finger table whose identifier is largest of those smaller than the target identifier. This algorithm converges quickly, and with high probability the hop count is O (log(N)) where N is the number of nodes in the network with a routing table size of log(N). In case of node failures, the performance is somewhat degraded. (Stoica, et al., 2001)

As a node joins the network, the network's routing information must be maintained. This is done by making sure that the successor information for each node is always correct and making sure that the correct node is responsible for the correct keys. To maintain efficient routing, the finger tables of all hosts should also be as current as possible, even though the routing will remain functional even if only the successors are known for each node. In addition to the finger table, for the purpose of simplifying joins and parts, nodes also maintain information about their direct predecessors. (Stoica, et al., 2001)

To join the network, a host first needs to find a node already in the network. Like CAN, Chord does not specify how this information is acquired but relies on some external mechanism. The host wishing to join generates its identifier by hashing its IP address and then asks the node already in the network to look up its predecessor and successor. The joining node can then ask its predecessor and successor for their finger tables and generate its own finger table based on these, as they are likely to be close to correct for it as well. (Stoica, et al., 2001)

After joining and initializing its own finger table, the new node must then be entered into those nodes' finger tables which precede the new node by at least 2^{i-1} and which ith finger

of the table is the successor of the new node. This is done by for each finger i in the new node's finger table the new node contacts node $n - 2^{i-1}$, where n is the identifier of the new node, and starts walking backwards from there until a node is found the *i*th finger of which precedes the new node. The number of nodes that need finger table updating is $O(\log N)$ and getting all of the necessary updates takes $O(\log^2 N)$ time where N is the total number of hosts in the network. As the complexity is logarithmic, the join operation can be said to scale well. After the new node has joined the network and the relevant finger tables have been updated, all that is needed to complete the join is to transfer the ownership of the relevant keys. (Stoica, et al., 2001)

The above algorithm may cause problems in the real world. As concurrent joins and both explicit and unannounced parts of nodes from the network can happen, it is possible that the finger tables are not always correct. In this case lookups may slow down, or in the extreme, if even the successor pointers are incorrect, may fail entirely. For such cases, a stabilization procedure is needed so the incorrect information is corrected so the lookup will work after being run again after a short while. This stabilization is realized with a periodically run operation in which a node requests its successor for its predecessor. In case the identifier of the successor's predecessor lies in between those of the requesting node and the successor, the node corrects its own successor information. As all nodes run this procedure intermittently, it is guaranteed that the network will eventually correct itself. (Stoica, et al., 2001)

A node failure must not disrupt ongoing queries in the network. Also, the failed node must be replaced by its successor in all of the finger tables it exists in. As the hosts notice the failure, they begin to update their finger tables by finding the successor of the failed node. To better facilitate this, in addition to finger tables and predecessor pointers, the nodes keep track of r nearest successors and use a similar procedure to the previously mentioned stabilization to keep this information up to date. It can be shown that with a successor list of length $O(\log N)$ where N is the number of nodes in the network, in an initially stable network where the probability of host failure is ½, the expected time to rebalance the network for each host is $O(\log N)$. (Stoica, et al., 2001)

While several implementations of Chord are available (Chordless, 2011), (Open Chord, 2011), (jDHTUQ, 2010), they are all written in Java which, while satisfying the requirement of running on Windows, Linux as well as OSX platforms, would likely require considerable work in order to be combined into the current C++ codebase client as well as producing considerable processing overhead. Additionally, none of the implementations are actively developed with the most recent update dating back over a year. Moreover, even though raw DHT implementations exist, it seems few applications actually take advantage of these or use their own implementations for any purposes. It seems therefore likely that some other solution might be able to provide a more confident solution. (Stoica, et al., 2001)

3.1.3. Pastry

Pastry (Rowstron & Druschel, 2001) is a totally decentralized, self-organizing peer-to-peer node mesh. Like CAN and Chord, Pastry is claimed to be robust, efficient and scalable. Additionally, it is able to take advantage of network locality in its message routing. Existing applications using Pastry include PAST (Druschel & Rowstron, 2001), a distributed data storage utility and SCRIBE (Castro, et al., 2002), a decentralized multicast infrastructure facilitating the creation of multicast groups and providing an efficient best-effort way for sending messages to members of those groups.

Like Chord, Pastry is based on the nodes arranged in a circular coordinate space. Each node has a 128-bit identifier ranging from 0 to $2^{128} - 1$. The identifier is assigned randomly as a node joins the network and should be generated so no collisions occur and so the identifiers are uniformly distributed across the identifier space. As is the case with other DHTs, the most central function of Pastry to route a message with a given a key to the node in the network whose identifier most closely matches the key. (Rowstron & Druschel, 2001)

Pastry has three control parameters, b and L, and M. All nodes store a routing table, a neighborhood set and a leaf set. b affects the amount of hosts kept in the routing table, L

affects the number of hosts kept in the leaf set and *M* the number of nodes kept in the neighborhood set. (Rowstron & Druschel, 2001)

The 128-bit key is divided into digits with each digit consisting of *b* bits. A routing table consists of $\log_{2^b} N$ rows each consisting of $2^b - 1$ entries where *N* is the number of nodes in the network. The *n*th row in the routing table contains the information of hosts in close proximity to the node with whom the node shares the first *n* digits and the *n*+1th digit is something else than that digit of the identifier of the node. If a node with a suitable identifier for a slot in the routing table is not found, a slot can be left empty. The choice of parameter *b* is a compromise. As *b* increases, the size of the routing table increases but the average number of hops decreases. (Rowstron & Druschel, 2001)

The neighborhood set stores the information of the *M* nodes closest to the current node according to a chosen proximity metric. While not normally used for routing purposes, it can be useful in maintaining the locality properties of the routing. A typical value for *M* is 2^b or $2 * 2^b$. (Rowstron & Druschel, 2001)

The leaf set stores the information of the nodes in the network closest to the current node in the identifier space. The information of *L*/2 of the nodes with the closest smaller and *L*/2 with the closest larger node identifiers is kept, and this information is used in routing in case the message is destined for one of these hosts. As with M, a typical value for *L* is 2^b or $2 * 2^b$. (Rowstron & Druschel, 2001)

As with CAN and Chord, the routing procedure is simple after the relevant tables have been constructed. When a message to be routed arrives, a node checks whether the message is destined for a node in the leaf set, and if so, forwards it to the appropriate node. If not, a node which shares a longer prefix with the key than the local node is searched and if one exists in the routing table, the message is forwarded to that node. If no such node can be found from the routing table or if the nodes are unreachable, the message is routed to a node with as long a prefix with the key as the current node but whose identifier is numerically closer to the key than that of the local node. By doinhg this, the routing converges with the

expected number of hops being $\log_{2^b} N$ and either finds a host for which the message is destined to or that no such host is up or exists. (Rowstron & Druschel, 2001)

If a node in the routing table has failed, it is detected when a message needing routing through that node arrives at the local node. The message is first routed through a different route after which the local node requests the routing table from another node in the same row of the routing table as the failed node. If all of the nodes with the right prefix have failed, the routing table of hosts from the next row are asked and then the next and so forth. This ensures that eventually it is very likely a suitable node is found if one exists. (Rowstron & Druschel, 2001)

To join the network, a prospective node must have information about a node already in the Pastry mesh. While not explicitly specified, it is suggested that an expanding ring search could be used, or some other method to make sure the node is as close to the host wishing to join as possible as per the defined proximity metric. After acquiring information about a nearby node in the network, the joining host generates and identifier for itself and sends a joint request to the node. This request is then routed to a node with the numerically closest identifier to that of the joining node. All nodes participating in the routing of this message, including the target node, send their routing, leaf, and neighbor tables to the joining node. The joining node parses this data, requests additional information from other nodes if it deems it necessary, and builds its own tables based on this knowledge. (Rowstron & Druschel, 2001)

The joining node builds its leaf set by copying it from the node with the closest identifier as they are likely to be very similar. It then initializes its neighbor set by copying it from the first host it contacted when joining as it is assumed this was in close proximity and is also likely to have much of the same information. The routing table is then updated by taking the appropriate values from each host along the route between the host with the closest proximity as per the metric and the host with the closest identifier, as each node identifier on the route has longer and longer shared prefix with the id of the joining node. The joining node then reports its state information to all of the nodes in its neighborhood set, leaf set and routing table and all those nodes update their respective information. The new node can then fully

participate in the functions of the network with the total number of messages exchanged having been $O(\log_{2^b} N)$. (Rowstron & Druschel, 2001)

A node in a pastry network is deemed to have failed when its neighbors can no longer communicate with it. When this happens, the neighbor nodes need to update their sets and tables. In order to replace an entry in the leaf set, a node contacts the node contacts the live node with the largest identifier if the identifier of the lost node was larger than its own, or if smaller, the node with the smallest identifier. The local node then requests the leaf set of that node. As the sets overlap, a suitable entry can then be found and copied to the local node's neighbor set. This procedure guarantees the neighbor sets of all hosts remain current unless $L/2$ hosts immediately adjacent all with larger or smaller identifiers fail at the same time. (Rowstron & Druschel, 2001)

The neighborhood set is kept current by all nodes periodically keeping contact with all hosts in their respective sets. If a connection cannot be established, the local node asks the other nodes in its neighborhood set for their neighborhood sets, evaluates the physical distance to any unknown nodes and updates its own set with the closest one. (Rowstron & Druschel, 2001)

The special feature of Pastry compared to many other peer-to-peer schemes is its inherent coherence to the locality of the underlying network. So long as there is some reliable defined metric available, such as round-trip time or the hop number from trace route, the routes in a Pastry network are likely be good in respect to that metric. As the routing tables are constructed after the join message from the routing tables lying in the route between a close node in the network and the closest node in the identifier space, the tables should contain nodes relatively close to the newly joined node as per the distance metric, i.e. the first row contains nodes close to the first contacted node, which are also close to the new node as well, the second close to the second, etc. (Rowstron & Druschel, 2001)

After the initialization, this locality still needs to be preserved to prevent it from failing when the network changes over a period of time. This balancing step consists of the node asking

all nodes in its routing table for their neighborhood sets and routing tables and updating any physically closer nodes to their appropriate places in its own routing table. This step and especially the trading of neighborhood sets ensure that changes in the network propagate so the locality properties of the network are preserved. (Rowstron & Druschel, 2001)

Because of the locality preserving nature of the construction of the routing tables, in Pastry the messages are always routed towards a node which is in relatively close proximity to the current node and share a longer prefix with the key (or in the case of a failed node, a node which is numerically closer). While this does not guarantee optimal routes, it is guaranteed that the routing does converge, and in the real world, results in relatively good performance. This also brings with it an additional benefit if redundancy is desired: in case a single key-value pair is replicated on multiple hosts by having them stored on all hosts in close proximity to the key in the identifier space, the messages are likely to end up to a node in relatively close proximity to the one which sent them. (Rowstron & Druschel, 2001)

Some problems arise from this routing and network building scheme. In case of routing problems with the underlying Internet Protocol (IP) causing parts of the Pastry network to become unreachable to each other, they will repair themselves to work as separate entities, a state which may then persist even after network connectivity is restored. Therefore, the protocol does require some additional method of peer discovery to ensure its functionality even the case of such failure. (Rowstron & Druschel, 2001)

An open source implementation of Pastry is available (FreePastry, 2009), but like the previously discussed chord implementations, it is written in Java. Also similarly, it boasts impressive plans “to provide a fully secure implementation that is suitable for a full-scale deployment in the Internet” but has not seen updates for years. What gives Pastry more confidence, though, is the fact that applications utilizing the technology do exist. (Rowstron & Druschel, 2001)

3.1.4. Tapestry

Tapestry (Zhao, et al., 2001) is a decentralized peer-to-peer overlay scheme, and like the previously introduced technologies, boasts scalability, robustness and self-organization. Like Pastry, it also attempts to take advantage of network locality according to some metric such as the number of network hops, latency or available bandwidth. Also like the previously discussed technologies, the most central feature of Tapestry is the location of data via key-value pairs through the implementation of a DHT.

The arrangement of nodes in Tapestry is very similar to that in Pastry. Node identifiers of *b* bits in length are derived for nodes via a cryptographic hash of some unique identifier. The identifiers are then split into digits of an arbitrary base. The routing is done by incrementally forwarding the messages towards a live node closest in network distance to the current node with a matching suffix, changing one digit at a time until the destination node is found. This ensures the maximum number of hops between any nodes is equal to the number of identifier digits assuming no routing anomalies exist. (Zhao, et al., 2001)

A root node is responsible for each key-value pair and is identified by the key hash. Each node containing the object publishes this information by contacting the object's root node, meaning the node responsible for the mapping. This information is also stored by nodes which store only the location of the closest copy of the object, allowing for quicker return of results to queries since not all queries have to be routed all the way to the root node. (Zhao, et al., 2001)

The routing is based on a neighbor map each node maintains. The neighbor map is arranged in levels based on matching suffixes with each level containing a number of closest nodes as per the distance metric matching the suffix of the level. In addition to a routing table, all nodes maintain a back pointer list containing the information of nodes whose neighbor the local node is which is used to generate the neighbor maps of new nodes joining the network. (Zhao, et al., 2001)

As faults are more likely as a distributed system grows larger, Tapestry aims to be an extremely resilient and robust by being able to resume effective functions by quickly detecting and adapting to problems in the network, such as node failures, connection outages and even corruption of the neighbor tables. (Zhao, et al., 2001)

The tapestry nodes detect node outages by two methods. TCP timeouts are used to detect a node becoming unresponsive, and each node periodically sends heartbeat packets over the User Datagram Protocol (UDP) to all hosts in its back pointer list to maintain awareness they are still available and functioning. Corrupted neighbor tables are detected by simply checking that the heartbeat messages arriving are indeed coming from hosts in the neighbor table. (Zhao, et al., 2001)

As the neighbor table contains the information of multiple hosts for the same path, if the closest one fails a fallback is very likely available. This makes for a quick recovery of functions when nodes fail. However, even if a neighbor is deemed to have failed, it is not removed from the neighbor table until a grace period of time has elapsed. This avoids having to perform a more costly join operation rather than just having kept its information available should the node return. If it can be assumed that the purpose of the network is such that a node leaving the network is rarely permanent, this is a sound approach. (Zhao, et al., 2001)

Tapestry networks are seldom saturated but rather more commonly sparse in the identifier space, and thus it is rare that a node identifier matching the key of an object exists. This problem is solved by a technique called surrogate routing, in which a surrogate node is assigned to those keys which lack a real root node. This scheme assumes that a node with the exact identifier exists and routes the requests for that key towards that identifier. When an empty entry in a neighbor table where a route should exists is found, the message is routed towards some deterministically determined host in the neighborhood of the target identifier. The routing ends when it reaches a host on whose neighbor map the only non-empty entry towards that identifier is the node itself. That node is then determined to be the surrogate node for the key. As a neighbor table entry can be empty only if there are no suitable hosts in the entire Tapestry network, it is guaranteed that this same host is always reached. (Zhao, et al., 2001)

Tapestry also proposes a redundancy scheme in which the key-value pairs are replicated on multiple nodes. This is done by adding a constant salt to them, e.g. a sequence of contiguous numbers, and replicating the pairs on the nodes which are responsible for the resulting new keys. This also enables Tapestry to send multiple queries simultaneously and choose the closest result as per the distance metric, resulting in additional locality benefits. (Zhao, et al., 2001)

A host wishing to join a Tapestry network first needs to obtain a knowledge of a close-by node already in the network. As with Pastry, this has to be done via some external method such as an expanding ring search or other bootstrap mechanism. The joining begins by the prospective node sending a join request with an identifier it has created to the known node. The new node then sends a message to itself through the known node, asking for each node on the route for their neighbor table and updating its own based on these. (Zhao, et al., 2001)

After finishing, the new node is likely to have a good approximation of an optimal neighbor table. The new node then sends a message to the node which is its surrogate and takes responsibility of those keys of the surrogate designated to new node. Next, the neighbors of the new node must be notified of its existence to update their own neighbor sets. This is done by traversing the back pointers of the former surrogate node of the new node as far back as surrogate routing was necessary and informing the hosts along the route of the new arrival after which the new node also informs all nodes in its neighbor set about its arrival. Now, the new node is a fully fledged member of the Tapestry network. The node then keeps its neighbor set updated by periodically measuring the distance to its neighbors as well as monitoring their heartbeats and adjusting its neighbor table accordingly. (Zhao, et al., 2001)

This method of joining the network is somewhat cumbersome which is why failed nodes are given a grace period before they are assumed gone and deleted from the neighbor tables of nodes. The deletion, however, is a simple procedure. If a node fails silently, it is simply removed from the neighbor tables after the grace period, and in case the node knows it is

leaving and not coming back, it can notify all its neighbors through its back pointers allowing them to remove it immediately. (Zhao, et al., 2001)

The root and surrogate node based location scheme can cause some problems in Tapestry. As hosts need to publish their available content, this can create significant bandwidth especially as the networks become very large. As the joining and balancing of the network is somewhat complex, it also seems that Tapestry is somewhat unsuitable for networks in which a lot of hosts are in a state of change at any given time. As this is a likely scenario in the context of device management, Tapestry appears less than suitable for the purpose. (Zhao, et al., 2001)

A dated java implementation (CURRENT Lab, 2006) and a somewhat less dated C implementation (CURRENT Lab, 2006) of Tapestry do exist and applications such as OceanStore (Rhea, et al., 2003), Bayeux (Zhuang, et al., 2001) and Mnemosyne (Hand & Roscoe, 2002) have been developed, the technology seems to have found little use outside of research and development.

3.1.5. Kademlia

Kademlia (Maymounkov & Mazières, 2002) is a DHT which provably maintains its function in an environment with faults, and is the basis of the DHTs used for locating peers by BitTorrent clients and implementations of which are used by various peer-to-peer applications. It aims to use as little resources as possible for maintaining the topology while maintaining quick operations even in the case of node failures.

Both the keys and the node identifiers in Kademlia are 160-bit-long binary strings. The network is modeled as a binary tree, as presented in Figure 7, with each node being a leaf in the tree with the position in the tree defined by the shortest unique prefix of the identifier. For each node, the binary tree is divided into a sequence of subtrees of diminishing size

which do not contain the node itself. All nodes keep in touch with at least one node in each of these subtrees to facilitate routing. This then makes it possible to reach any node in the network in a number of hops which is logarithmically proportional to the amount of nodes in the network. (Maymounkov & Mazières, 2002)

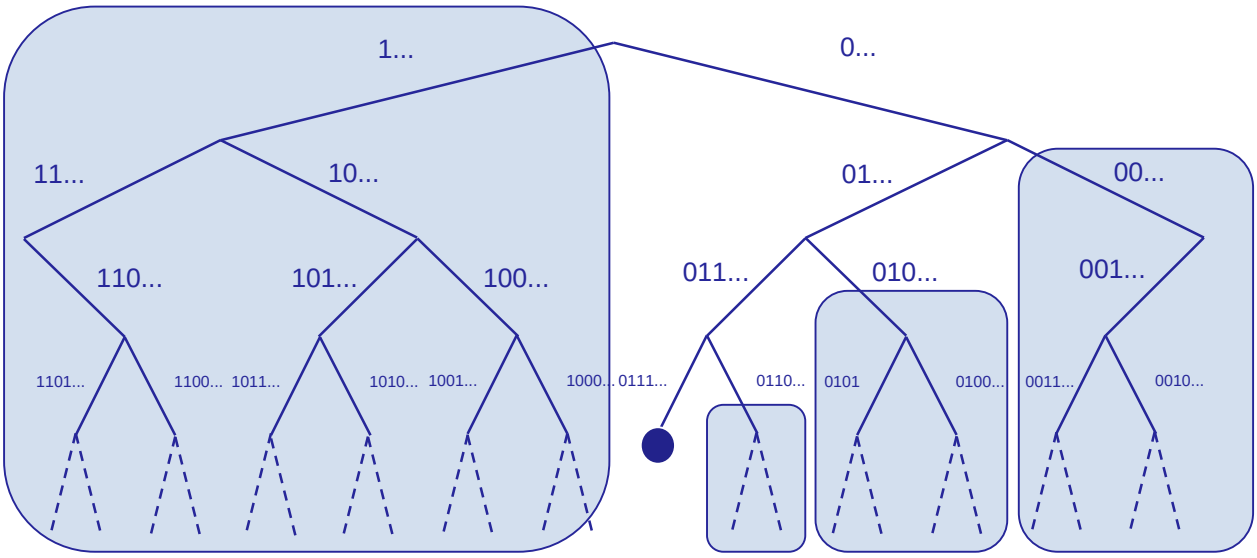

Figure 7 - Kademlia topology

The network health is maintained by nodes keeping other nodes in records called k-buckets. Each hosts keeps 160 k-buckets, each one containing *k* nodes between the distances 2^i and 2^{i+1} from the nodes themselves where $0 \leq i < 160$ and typically $k = 20$. The k-buckets are arranged as lists with the most recently seen node at the tail end and the least recently at the head. Each time any message is received from any node, the node is either inserted or bumped to the tail end of its k-bucket. If the k-bucket is full, however, the node sends a ping message to check whether the node at the head end of the k-bucket is live, and if it is, it is instead bumped to the tail end and the information of the new node discarded. This provides reliability both since it is likely that nodes having remained up for a long time will continue being so and since it's impossible to fill the buckets with unusable nodes by flooding the network. The buckets are normally maintained by requests not directly related to their health, but in case some bucket receives no requests for an hour, a node randomly picks an id from its range and performs a node lookup for it. (Maymounkov & Mazières, 2002)

Kademlia works with four operations: *ping* simply checks whether a given host is live, *store* stores a key-value pair in the network, *find_node* finds a node associated with a given ID and *find_value* finds the value associated with a given key. A *find_node* is responded with the information of the *k* nodes the receiving node knows that are the closest to the target id. The *find_value* request differs from this only in that if it was sent to the node storing the value, only the value is returned. The distance metric used in Kademlia is the exclusive or (XOR) of the identifiers or keys interpreted as an integer. (Maymounkov & Mazières, 2002)

Node and key lookups are done iteratively, meaning all queries are returned to the original node which continues querying other nodes based on them. Parallelism is also exploited. the parameter α controls how many nodes are queried simultaneously with larger numbers resulting in faster queries as well as more bandwidth usage. A typical value for α is 3. As the queried nodes return results, the results are then arranged according to their proximity to the target and the closest hosts queried, continuing the process with their results until the search converges. This is also done before running the store operation, as reliability in the network is increased by storing values on the k hosts closest to the key. After locating these nodes, a node sends each one a store request. When using *find_value*, the queries are immediately terminated when any node returns the value, and for additional caching purposes, a store request for the key is sent to the closest node which did not respond with the value. The expected number of iterations that need to be performed for an arbitrary lookup is $\log(n)$. (Maymounkov & Mazières, 2002)

The key-value pairs are stored on *k* closest nodes to the key as per the XOR metric. The larger the chosen value for *k*, the more resilient the data storage against lost nodes but the amount of data needed to keep the key-value pairs grows in direct proportion. The original specification calls for *k* to be chosen so it is very unlikely for *k* randomly picked nodes to fail within an hour of each other. (Maymounkov & Mazières, 2002)

When first joining the network, the new node contains only a single k-bucket as the only host it knows is itself. After getting knowledge of some other node in the network, the joining

node inserts it into the appropriate k-bucket and then performs a node lookup for its own node identifier. After this, the new node runs a random node lookup into each of its k-buckets further away than its closest neighbor, populating its own and inserting itself into the k-buckets of other nodes. A special case is the k-bucket which includes the node identifier of the node itself. If that k-bucket is full and an insertion to it comes to pass, it is split in half into two k-buckets, the contents split between them and the insertion retried. (Maymounkov & Mazières, 2002)

As Kademlia based DHTs form a part of the core of such peer-to-peer applications as BitTorrent and eDonkey2000, it can be stated with some confidence that it is a reliable DHT solution. There also exists a dated C library (Michelangeli & Jalkanen, 2006) released under the GPL license (Free Software Foundation, 2007) and a more recent C++ implementation called maidsafe-dht (MaidSafe, 2012) under the New BSD license (Open Source Initiative, 2012) boasting such features as NAT traversal and security as well as ability to work on windows, OSX and Linux making it a lucrative adaptation possibility.

3.2. Content sharing

Content sharing is a very popular application of peer-to-peer networks, and indeed a very popular use of the internet. From 2007 to 2009, peer-to-peer file sharing traffic was worldwide the largest single use of the internet, encompassing between 40 and 85 percent of all internet traffic depending on the time and location (Schulze & Mochalski, 2007) (Schulze & Mochalski, 2009). This section discusses some of the historically and currently popular solutions for internet-scale peer-to-peer file sharing.

3.2.1. Napster

Napster was a service which during its operation enabled users to join a peer-to-peer network in order to share music files with each other. With the network beginning operation in 1999 and ending with a court order in early 2001 (Liebowitz, 2006), Napster enjoyed great

popularity, claiming over 70 million registered and over 1.5 million simultaneous users at its peak popularity (Fox, 2001). Succeeding the order to shut down the network, the company went bankrupt a year later after a legal battle with copyright infringement as the central issue (CNN, 2002).

While all the file transfers happened directly between network nodes in a peer-to-peer manner, Napster was very much a centralized system. The central server hosted the entire directory of nodes and their file lists to support a search function and offered an Internet Relay Chat (IRC) like chat function for the users. Apart from the file transfers, no decentralized scheme existed for content or peer discovery, and as shown by the court order, this provided a single point of failure able to shut down the entire network. Also, the system was very specific to sharing audio files, specifically music, with the search and indexing tailored to searching by artist and song names and filtering with such variables as bitrate. The system also allowed users to report the speed of their connection to let other users know whether they might be a good host to download from. The reported bandwidth, however, was not verified in any way and it was commonplace that the real and reported speeds were quite mismatching. Another way to select a likely good peer was called ping, the round-trip time for a small packet to the host, which was another variable the central server kept track of.

While a revolutionary service in its time, due to the nature of the content shared by its users, it encountered legal trouble from copyright holders. Due to the technical solutions used to build the network, it proved easy to shut down. It was, however, the application that brought file sharing to the forefront of public consciousness and popularized the concept of peer-to-peer networks.

3.2.2. Gnutella

With the first client released in early 2000, Gnutella (Klingberg & Manfredi, 2002) was the earliest fully decentralized peer-to-peer protocol. It was built for the purpose of sharing files between users on the internet and developed by Nullsoft. As Napster was already suffering

from legal fallout, many users were looking for alternate file sharing solutions and Gnutella succeeded in filling this void, making it an immediate success even though AOL, the parent company of Nullsoft, stopped its distribution almost immediately (CNN Tech, 2000). After the AOL pullout, the protocol was quickly reverse engineered and third party implementations began to appear with such programs as gtk-gnutella (Grossel, 2000) and LimeWire (Lime Wire LLC, 2000) released within months. Gnutella has since proven pervasive, regularly hitting over 3 million concurrent users in early 2006 (Stutzbach, et al., 2007) and while losing market share to eDonkey and especially BitTorrent, in 2009, almost ten years after its conception, Gnutella still caused a sizeable portion of all peer-to-peer traffic on the internet (Schulze & Mochalski, 2009).

Gnutella nodes, called servents, join the network by contacting an existing servent. As all the other techniques presented, Gnutella relies on an external method for bootstrapping this joining process if no servents are locally cached or none of the cached hosts respond. One such method is Gnutella Web Caching System (GWebCache) (Dämpfling, 2003), a distributed host caching system which provides interfaces for clients to both download and upload node lists. While the original specification and implementation are dated in 2003 (Dämpfling, 2003), the usefulness of the system has spawned many others with caches still running as of July 2012 (Dan-To Services, 2012) (Link, 2012) (机电工程师, 2012).

All Gnutella messages start with a header containing a globally unique identifier (GUID), payload type, time to live (TTL), Hop count and payload length. The GUID is a 16 bytes long string with *0xff* stored in byte 8 to signify a modern servent and *0x00* in byte 15 which is reserved for future use. The payload type is one byte with possible values of Ping, Pong, Bye, Push, Query and Query Hit. The one byte long TTL value signifies the number of hops the packet should be routed in the overlay, each servent the message passes through decreses this value by one and once it reaches zero, the message is dropped. The one-byte hops field counts the number of hops the message has transited, each servent routing the message increases this by one. Finally, the four-byte payload length specifies how many bytes the rest of the message, containing nothing but the payload, is. (Klingberg & Manfredi, 2002)

When a suitable host is found, the joining host sends a PING message to the node already in the network, using TCP. The default listening port is 6346 but other ports may be used. The Ping message contains only the header with the exception of optionally containing a Gnutella Generic Extension Protocol (GGEP) block containing custom extensions to the protocol not necessarily supported by all clients as a payload. The target servent responds with a Pong message containing its port number, IP address, number of shared files, amount of shared data in kB and optionally, a GGEP block. Additionally it is valid for the servent to respond to a Ping message with multiple Pongs. This facilitates quick dissemination of host information to new hosts in the network as the new servent does not necessarily have to explicitly send Ping messages with a larger TTL to probe for the network. Such Ping messages with a TTL of 2 and hops of 0 are known as Crawler Pings and are responded with Pongs with the information of all the hosts it is connected to. This can be implemented either through generating the Pong messages based on its host cache, or generating Pings for all of the hosts and forwarding the Pongs to the host having sent the crawl. (Klingberg & Manfredi, 2002)

A Query message is used for locating content on the Gnutella network by keywords. It may be generated either by a user making an initial query, or automatically by a servent to locate more sources for a file. After the header, a Query message has two fields, Minimum Speed and Search Criteria, with an optional extension block. The two-byte Minimum Speed specifies that a servent should not respond with a Query Hit even if the query otherwise matches if its available upstream bandwidth is less than the indicated number in kbps. From the third byte onward, the Search Criteria forms the rest of the message assuming no extensions are used. The Search Criteria contains a string of keywords that should be encoded in 7-bit ASCII or if the 8^{th} bit is used, there should be either a GGEP extension block in the message specifying the encoding or the recipient should make a guess of encoding or encodings against which to try and match the query. The query should only match if all keywords are found in any file name and in such case a Query Hit message is sent back to the sender and if the TTL has not expired, the Query message routed forwards to all the neighbors of the servent except the one it came from. Since this may cause the same Queries to arrive to the same servent through multiple routes, a servent should cache message

IDs and drop any messages already seen. As Gnutella simply floods the searches to all neighbors of all encountered nodes until the TTL expires, the protocol specifies that Queries above 256 bytes in size can be dropped and those above 4kB should be dropped. (Klingberg & Manfredi, 2002)

A Query Hit message is generated when a search matches the Search Criteria specified in the Query message. Its mandatory fields are Number of Hits, Port, IP Address, Speed, Result Set and Server Identifier. In addition, the protocol specifies a recommended extra block and an optional Private Data block. The two bytes long Number of Hits specifies the number of results in the Result Set. The Port field consists of the next two bytes and specifies the Port on which the servent can receive incoming file requests and the four-byte IP Address field specifies the IPv4 address of the servent. The Result Set is a group of four fields repeated Number of Hits times. The first four contain the File Index, a number assigned by the servent uniquely identifying that file on that host and the next four the file size in bytes. Following these is the File Name field of arbitrary length, containing the file name and terminated by a null. Additionally, a null terminated extension block is mandatory and if no extensions are specified, a double null. After the Result Set, the Query Hit message may end in the Servent Identifier or contain the recommended extra block, which consists of fields Vendor Core, Open Data Size, Open Data. The first four bytes contain a four character vendor specific code identifying the client vendor. The next byte contains the length of the following Open Data field and can be either 2 or 4, depending on the features supported by the client. The Open Data field then contains that number of bytes worth of flags specifying things such as whether the servent has successfully uploaded at least one file, whether it currently has any open upload slots available or if it can or cannot accept incoming TCP connections. The optional Private Data field may then contain arbitrary vendor specific data or, if specified by a flag in the Open Data field, a GGEP extension block. Finally, the last 16 bytes of the message are reserved for the Servent Identifier. (Klingberg & Manfredi, 2002)

For transferring files, Gnutella clients implement a HTTP server. Once a file is found and the user decides to start downloading, the user's client tries to send a HTTP GET request to the target servent with file index and file name as the arguments and specifying a Range field in the header to specify which bytes of the file it is requesting, thus facilitating getting only

parts of files and getting different parts of files from different hosts. In case the servent with the desired file is not reachable through incoming HTTP, the Gnutella protocol specifies a Push request the fields of which are Servent Identifier, File Index, IP Address, Port, and optionally, a GGEP extension block. After the 16-byte Servent Identifier, the four-byte File Index is the same as specified by the servent in its previous Query Hit message. The four-byte IP address and two-byte Port fields then contain the IPv4 address and TCP port to which the servent should attempt to Push the file. Additionally, the GGEP extension block can be then follow for arbitrary purposes. (Klingberg & Manfredi, 2002)

Finally, a bye message can be, but does not have to be used when a servent wants to get disconnected from one or more of its neighbors and specify a reason, e.g. when leaving the network. The Bye message contains a payload consisting a Code and a Description field, and a header with the Payload Type set to the corresponding value and, a TTL of 1 and a Hops of 0. The two-byte Code field contains the error code as per the Enhanced Mail System Status Codes range specification (Vaudreuil, 1996) with codes 2** specifying normal behavior, 4** a user error and 5** an internal error. The following bytes then contain a null terminated Description string. After receiving a Bye message, a servent must close the connection. The protocol also specifies that after sending a Bye message the sender should wait for some seconds for the remote host to receive the message and close the connection before closing it itself. Alternatively, it is legal for a servent to just close any connections without sending a Bye. (Klingberg & Manfredi, 2002)

Depending on the message, they can be routed in various ways by the servents in the network. A Ping message is usually sent only to a single host and not routed, but a Crawler Ping may be routed up to one hop. A Push message is only routed towards its recipient, as are Query Hit messages. A Bye request should never propagate beyond its first recipient. Queries, however, are flooded across the network, as the protocol specifies that each host receiving a Query must forward it to all its neighbors until the TTL, decreased by each host in the routing chain, reaches zero or if the servent has already forwarded the same Query message in which case it is dropped. This makes searching for content in the Gnutella network somewhat cumbersome. Even if the Queries are dropped by servents that have seen them before and the packets are very small, flooding the entire network generates

unnecessary traffic. Additionally, as TTL is the only scoping method and a large TTL is frowned upon, there are no guarantees that a Query Hit is produced even if the file does exist on a servent on the network. (Klingberg & Manfredi, 2002)

While the specification provides the aforementioned definitions on the protocol itself, it also provides non-mandatory recommendations on how the protocol should be used and how the servents should behave in order to make and keep the network functioning more effectively.

As no guarantees can be made about the connection speeds and processing power of servents, Gnutella clients should implement flow control. The specification allows for a simple solution of dropping packets and closing connections in case of overload, but this would likely worsen the reliability of the routing in the network. To increase reliability, a flow control method is suggested in which each servent implements a buffer for outgoing messages, working in simple FIFO mode unless it becomes filled up to 50% after which the servent enters the Flow Control mode in which all incoming Queries are dropped and other messages in the buffer are routed according to their priority. The priorities are then calculated by making messages propagating through broadcasting less important the more hops they have traveled, Query Hits the more important the more hops they have traveled, and through categorizing by message type in a descending order from most to least important: Push, Query Hit, Pong, Query, Ping. Also, if the servent wants to send a Bye message, the buffer should be clear of other messages going to the target of the Bye before sending it. (Chawathe, et al., 2003)

Ultrapeers (Singla & Rohrs, 2001) are a concept that make Gnutella somewhat less of an egalitarian peer-to-peer system but move it towards a two-tier structure similar to the supernode system of Skype. Proposed to increase the scalability of the network, this system has the servents split into two classes, Ultrapeers and Leaves. A Leaf node only has a small number, typically up to ten connections to Ultrapeers while an Ultrapeer is normally connected up to a 100 (Ilie, et al., 2004). As with the supernodes in Skype, a Gnutella servent may become an Ultrapeer if it supports the additionally specified headers as well as the Query Routing Protocol (QRP) (Rohrs, 2001) and meets the requirements of not being

firewalled or NATted, having a suitable operating system, having enough bandwidth and system resources available and being likely to have a relatively high uptime.

Ultrapeer eligibility is determined by the servent itself and is signaled through an additional X-Ultrapeer header field in Ping messages. If X-Ultrapeer is set to True, the signal is that the servent connecting is or wishes to be an Ultrapeer, and if False, cannot or will not. When a new host joins the network, the servent it first contacts can respond in several ways. In case the connecting host wants to be a leaf node and contacts another leaf node, it only gets a response of Ultrapeer addresses to try and contact before closing the connection, unless the node in question knows of no Ultrapeers in which case it accepts the connection and starts routing the messages as if in a Gnutella network with no Ultrapeer capability. When the host wishing to be a leaf node contacts an Ultrapeer, it gets a list of Ultrapeers and normal servents to try and contact in case the connection is lost. The newly connected node then closes its connections to other hosts and sends their information to the Ultrapeer in the form of a QRP routing table. In the case of an Ultrapeer capable host connecting to an Ultrapeer, the Ping functions exactly as in normal Gnutella, except in the case of the Ping recipient replying with a X-Ultrapeer-Needed header with a value of False in which case the Pinging node becomes a leaf node subservient to the Ultrapeer in question and in case it has managed leaf nodes sends its QRP routing table to its new Ultrapeer. (Singla & Rohrs, 2001)

The QRP routing protocol along with the Ultrapeer scheme aims to make Gnutella more scalable by reducing the amount of bandwidth needed by Queries by trying to stop the Queries from being uniformly broadcasted across the network by stopping the propagation of Queries to hosts who are known to be unable to produce a Query Hit (Rohrs, 2001). Its basic functionality is similar to a DHT, working by the servents hashing the keywords in the file names of the files they are sharing and keeping their neighbors up to date of this information. This information is shared through QRP tables, hash tables of 65kb of size with a binary value of 1 signifying a hit and 0 signifying a miss. Hashes map to indices on the hash table and in case a 0 is found, it is known that the Query will not produce a Query Hit from that servent and will not be routed there. Even if a 1 does not guarantee a hit, this is still likely to reduce the amount of data that needs to be transferred. When added to the Ultrapeer functionality so that Ultrapeers combines in its QRP table all the leaf nodes it

manages are sharing, routes in the network are likely be shorter than those in conventional Gnutella and fewer Query messages are likely to be generated.

While Gnutella was first devised more than ten years ago, the protocol is still relevant and indeed a time proven workhorse of the peer-to-peer world. The supreme popularity of BitTorrent has overshadowed that of other file sharing solutions but the techniques used in Gnutella have not all been surpassed and still offer insight into building a robust network.

3.2.3. BitTorrent

BitTorrent (Cohen, 2008) is a very popular if not the most popular peer-to-peer protocol currently in existence. It is used for content sharing. In 2008-2009, depending on the geographical area, peer-to-peer traffic encompassed between close to half and almost 70% of all internet traffic, with BitTorrent being the most popular single protocol in all but South America (Schulze & Mochalski, 2009). Even though BitTorrent shares some of the centralized aspects of Napster, notorious BitTorrent search engines such as The Pirate Bay have been able to keep operating despite tremendous legal and political pressure through convenient local laws and community support. The technology is also more pervasive than that of Napster and will be very hard to eradicate through centralized action.

A very basic BitTorrent system consists of three pieces. A .torrent metainfo file contains, as the name suggests, meta information about the files it describes. This information inclues suggested file names, piece length signifying the number of bytes each piece of the download is split into, the Uniform Resource Locator (URL) of the tracker, file sizes, paths, etc. (Cohen, 2008)

Peer discovery in BitTorrent is done either by the tracker, a server keeping track of the metainfo files and the nodes associated with the content, or through a DHT. The tracker server was the original method, but later DHT was implemented in order to avoid the tracker

being the single point of failure by distributing the node discovery to the nodes themselves, effectively making the network function as the tracker.

BitTorrent provides no authentication directly – anyone able to find peers on the network can contact them and request files from them. Access controlled BitTorrent networks exist, however, but the method of authentication is rather more indirect. Usually such a system consist of an access controlled web server for content searching and distributing the metainfo files, each of which contains a unique passkey passed onto the tracker as the BitTorrent client contacts it for authentication. If a client lacks or has an unknown passkey, instead of a node list, it will receive nothing or an error. The BitTorrent specification also allows for disabling the use of DHT node searching for any given .metafile, and while client programs don't have to respect this, client programs which do not are often banned from private trackers. Nothing prevents custom clients from sending fake version information to a tracker but it is currently unknown whether such clients exists and it can be said with high probability that they are not commonplace.

BitTorrent clients have historically implemented multiple DHTs. Azureus (Vuze, Inc., 2012) was the first to implement one which was and is not supported as a part of the protocol. Later, the official BitTorrent protocol adopted a similar system for distributed peer exchange known as Mainline DHT (Loewenstern, 2008). It is incompatible with the Azureus implemented one, but supported by many popular client programs. Both of the BitTorrent DHTs are based on the Kademlia DHT.

When a BitTorrent client wants to download a file from the network, it first needs a metainfo file. After getting this file through some outside delivery method, the official specification suggesting a web server for content searching and metainfo delivery, the client contacts the tracker for a list of nodes who have relevant, or, in the case of a trackerless metainfo, searches the DHT for such nodes. Like all the previously reviewed technologies, the Mainline DHT requires that some peer already connected to the network is known in order to bootstrap joining of the current node to the network. The method for such searching is not specified but an expanding ring search might be a good option, or, due to the pervasiveness of

BitTorrent in the internet, good performance might be attained even through trying to contact random IP addresses until a successful connection is made.

After getting a list of nodes, the client starts to both download the files and upload them to other nodes on the network. The files in BitTorrent have been arranged to pieces defined in the metafile, so files do not have to be downloaded as a single large file but rather a larger number of small portions which are then arranged into the correct order by the client program. This allows for easy distribution of bandwidth among a large swarm with the added benefit of no single slow connection being likely to dramatically slow down the entire transfer as well as enabling the client to start sharing its own upload bandwidth as soon as a single piece of the download has completed. While the original specification calls for downloading the file pieces in a random order, some clients support a streaming feature implemented by downloading the file pieces sequentially in order to facilitate for example starting to watch a video before the file has downloaded in its entirety. This somewhat skews the data availability especially in the case of new content where instead of propagating equally onto the network to allow for quick distribution, all clients want to download the same pieces which are available only on a limited number of hosts. Though especially with popular content, this is probably unlikely to cause problems in the real world.

BitTorrent offers very limited solutions to firewall or NAT traversal. Hosts behind address translators cannot be contacted from the outside, but the clients inside the NAT try to circumvent this by periodically refreshing their node lists and contacting any nodes in need of content the client has or who have such content that the client wants.

3.3. Botnets

Botnets are platforms for distributed computing, often propagated through exploitation of operating system or application vulnerabilities or social engineering (Bailey, et al., 2009) and used for malicious purposes (Abu Rajab, et al., 2006). Botnets provide a greatly relevant research topic as essentially the system specified in this thesis is essentially just that – a

botnet overseeing and initiating arbitrary behavior of hosts, albeit with the difference to most botnets that the bot master is a benign entity.

Botnets consist of bots, i.e. the infected hosts running the bot program, and some kind of a Command and Control (C&C) channel to reach them. The most commonly used C&C method has been IRC (Cooke & Jahanian, 2005) (Barford & Yegneswaran, 2007), but HTTP has also been discovered to have been used (Abu Rajab, et al., 2006) and most recently peer-to-peer solutions have been emerging (Abu Rajab, et al., 2006) (Wang, et al., 2010). While commonly used for malicious purposes such as denial-of-service attacks, the properties of good botnets would be very similar to those required of a good solution for device management.

The recent Advanced Hybrid Peer-to-Peer Botnet (Wang, et al., 2010) presents a particularly interesting peer-to-peer oriented architecture for a botnet. Most botnets have usually had some kind of central C&C system, commonly a hardcoded IRC server for controlling the botnet, but as opposed to this, the paper proposes a model of decentralization which claims to be very robust to outside attacks.

The communications architecture of the scheme resembles those of both Gnutella and Skype. It is split into two tiers, regular bots, similar to those of the nodes in Skype or the regular Servents of Gnutella. Additionally, there is a higher level structure consisting of hosts called Servents, and like the supernodes of Skype and Gnutella Ultrapeers, are elected from a population with public IP addresses with good connectivity. Any of the Servents can then be contacted by any of the bots and any bot commanded by any Servent. (Wang, et al., 2010)

As this scheme does not have predefined command routing structures, authentication of command becomes particularly relevant. The authors propose that public key encryption is used so that a public key is distributed with the bot program and everything signed with the private counterpart is trusted by the bots, a scheme similar to that used in Skype. (Wang, et al., 2010)

For encryption, the paper proposes symmetric key encryption with keys specific to the

Servents. Thus, if a single Servent is captured, the encryption is only compromised between that servent and the bots it manages and not the whole network, limiting the exposure. (Wang, et al., 2010)

Additionally, unlike most of the peer-to-peer schemes previously discussed, this one requires no bootstrapping procedure. Instead, it is proposed that a list of Servents is propagated through sending one along with each infection, and in the case of reinfection, probabilistically replaces some of the Servents in the bot's Servent list with new ones. This makes sense as one of the goals of the botnet is to minimize exposure in case of a compromised host and thus no single node ever contains a full Servent list but only parts of it. Reinfection also provides a way to update Servent lists with the additional benefit of making it hard if not impossible to generate an infection route even from the lists of a large number of compromised bots. However, the additional traffic generated by reinfections could itself make the botnet more detectable. (Wang, et al., 2010)

Through simulations, the authors argue that for a large botnet, even with 70% of the Servents removed, 95% of the botnet will still remain connected. They also argue that for the same botnet of 1000 Servents, as the size of the Servent list increases to 100 hosts, almost all bots will remain connected even if the best connected 95% of the Servents are removed, making the system quite robust. (Wang, et al., 2010)

The proposed architecture seems quite robust and secure and while not explored here due to its small relevance to this thesis, resistant to detection, and removes the single point of failure of the C&C server or servers by distributing the responsibility to the elected Servents. This requires additional security, the specification of which is of particular interest as the use case is quite similar. (Wang, et al., 2010)

3.4. Skype

Skype is a popular internet telephony service currently owned and developed by Microsoft. While Skype is not open source and has not published its protocol, some analyses (Baset &

Schulzrinne, 2004) (Caizzone, et al., 2008) (De Cicco, et al., 2007) have been performed on how it works. In addition to voice and video calls, Skype enables users to chat with text as well as send files to each other. Many things known about the inner working of the program are, however, known only based on high level comments made by representatives of the company in interviews or other public occasions.

Skype is essentially a two-tier peer-to-peer network split into nodes and supernodes. A node is simply any computer with the Skype program installed. A supernode performs some additional functions such as routing calls and being the middle man when nodes need to traverse firewalls and address translators (Desclaux & Kortchinsky, 2006). Formerly, any node with sufficient resources and reachability could become a supernode, but recently, Skype was changed so all supernodes are now computers running a version of Linux hosted by Microsoft (Goodin, 2012). Additionally, the Skype network contains a login server (Baset & Schulzrinne, 2004) or servers (Perényi, et al., 2007) whose sole function is user authentication to allow for a Skype client to join the network using the credentials of a certain user (Baset & Schulzrinne, 2004). This is done by verifying a user name and a password. In case a password is forgotten, Skype also offers a possibility to reset it using access to an e-mail address as a backup proof of identity (Skype, 2012).

Normally, the nodes in the Skype network communicate with each other directly. However, if a direct connection cannot be trivially established, some additional processing is needed. In such a case, a supernode reachable by both of the nodes will act as a mediator between the two. Both of the nodes will send UDP packets to the supernode, which will then respond, giving both the other's remote IP address and UDP port, which might then be reachable from the outside internet. The hosts then try to establish a connection among themselves, and if successful, the supernode has to do no more. However, in case the nodes cannot manage to negotiate a connection at all, a supernode can also act directly as a call router, forwarding the packets from the hosts to each other.

Skype is renowned for its ability to function almost anywhere and automatically traverse through a wide variety of address translators and firewalls. However, because of this and the closed source nature of the program, it is viewed as a security threat by some as there is no way to actually verify what information is transmits over the encrypted connection. Skype,

however, makes no attempts to disguise its protocol, it only encrypts the content. Even some nation states, such as China have tried to limit the use of Skype due to security concerns, while some, such as Ethiopia, have categorically banned all VoIP services, thus including Skype. Despite doubts, Skype claims to be a secure, using RSA for key negotiation and AES for symmetric key encryption between nodes for both text and voice communication (Skype, 2012). Skype has, however, explicitly refused to comment on whether it is possible for them to decrypt the communications between Skype nodes and listen in. (Sauer, 2007).

Skype also appears to be very scalable. According to recent news approximately 10000 dedicated supernodes exist (Goodin, 2012) in a network which has seen over 40 million simultaneous users (Caukin, 2012). The supernodes, however, used to consist mostly of normal Skype clients running on hosts with sufficient reachability and spare processing power could become supernodes (Desclaux & Kortchinsky, 2006). In 2006, the number of supernodes in the network was around 20000 (Desclaux & Kortchinsky, 2006) while the number of simultaneous clients in the entire network peaked at over 5 million for the first time in late January (Jaanus, 2006). In 2006, a supernode could use a maximum of 5 kilobytes per second (kBps) of bandwidth for signaling purposes, and working as a relay, a maximum of 3 kBps for file transfers, 4kBps for voice calls and 10kBps for video calls (Skype Limited, 2006). As the bitrate requirements of Skype currently vary from 30 kilobits per second (kbps) minimum for voice calls to 8Mbps (megabits per second) for large video conferences (Skype, 2012), it may be that the requirements for the supernodes have also increased.

The Skype binary contains 14 hardcoded RSA moduli and the client trusts any messages signed with these. They are used in the login phase to ensure that the login server the client contacts is authentic. The login server and the client also use a shared secret for symmetric encryption, the hash of the user information. Additionally, the binary contains a version dependent list of 200 supernodes. (Desclaux & Kortchinsky, 2006)

To facilitate authentication, the client first generates a pair of RSA keys which are later used to identify the user either for the duration of the session or for a longer period if the user

chooses the client option to remember the credentials. A symmetric encryption key is also generated. The client then encrypts the hashed username and password along with the symmetric key using one of the trusted moduli and sends them to a login server. If the hash of the password matches that stored in the login database, the user is allowed to authenticate. The Skype name of the user and its associated public key are then distributed to supernodes.

When a user wants to call another, the client starts by signaling a supernode that it wants to call the remote user. The clients running with this user's credentials are then located by the supernode and the information about the incoming call routed to them. As the remote user picks the call up, the clients try to negotiate a connection between them. If a UDP or TCP connection between the IP addresses cannot be directly established, a supernode is used to try and negotiate the connection. While Skype does not explicitly disclose the methods used for this, it is likely they use some form of UDP or TCP hole punching is used. It is also known that in case other techniques don't work, Skype uses supernodes for relaying data between hosts.

3.5. Discussion

Many lines of solving exist for the many challenges in the peer-to-peer world, and many development efforts seem to have produced similar solutions to similar problems, yielding concepts such as the DHT and Ultrapeers or supernodes. While the emergence of such solutions have certainly not been wholly independent, a case may with good confidence be made for some which are likely to be good for certain purposes.

As it was discovered with the growth of the Gnutella network, the simple flooded Query requests quickly created congestion problems (Sripanidkulchai, 2001) as the load on each node grew in linear proportion to the number Queries (Chawathe, et al., 2003) or even faster (Portmann, et al., 2001). Many solutions were proposed which eventually developed into Ultrapeers (Singla & Rohrs, 2001) and QRP (Rohrs, 2001) which while keeping the network independent from centrally managed servers, moved it towards a hierarchical structure while

also greatly increasing its scalability as can be seen from the traffic moving from being only 36% Queries and the rest overhead to 92% Queries between November 2000 and June 2001 (Ripeanu, et al., 2002). As QRP is likely to have also diminished the number of propagated Queries across the network and as Gnutella is still used today, over ten years since its inception, it can be argued that the system is quite suitable for its purpose.

The currently most popular peer-to-peer network (Schulze & Mochalski, 2009), BitTorrent (Cohen, 2008), was originally, and in some ways still is far more centralized, utilizing central servers, including the notorious Pirate Bay, for both content searching and peer exchange. Thus, it would by intuition and as shown by the case of Napster (CNN, 2002) be more vulnerable to attacks, but it has still managed not only to rise to become the most popular peer-to-peer protocol but to maintain its popularity. Additionally, as DHT support has been added to the protocol (Loewenstern, 2008), its dependence on at least the tracker servers has somewhat diminished. The protocol itself, however, does not yet offer any other way except centralized databases or metadata files for content searches, making the BitTorrent network still more vulnerable to attacks than for example Gnutella.

Skype (Skype, 2012), the internet telephony network with a record of over 40 million concurrent users (Caukin, 2012), is a peer-to-peer application known for its ability to work almost anywhere with no special configuring despite firewalls or NATs so long as an internet connection is available. Like Gnutella, it is based on a tiered topology of nodes and supernodes, the latter of which act as routers and connection mediators for nodes. Additionally, Skype employs a network of login servers (Desclaux & Kortchinsky, 2006) to store and verify user credentials both to limit the access to the network as well as authenticate the identities of the users.

The more theoretical approaches discussed, CAN (Ratnasamy, et al., 2001), Chord (Stoica, et al., 2001), Pastry (Rowstron & Druschel, 2001), Tapestry (Zhao, et al., 2001) and Kademlia (Maymounkov & Mazières, 2002). All are centered on solving the problem of locating content on a distributed network, each solving the problem via a different variant of the DHT and each having a different emphasis on the properties of the networks they create. CAN simply seems to provide a no-frills solution for building a scalable network based on

a Cartesian coordinate system. Chord, Pastry and Tapestry are based on a simple circular bit string coordinate systems, each choosing a slightly different scheme to achieve efficient routing and small routing tables while trying to remain resilient in case of node failures. Pastry and Tapestry additionally try to take advantage of distance metrics to exploit network locality to try and match the overlay topology to that of the underlying physical structure. The topology of Kademlia is a binary tree and it relies on iterative routing as opposed to the recursive schemes of the other options. It also takes into account the likelihood of previously reliable nodes remaining so in the future, and slowly forms its network to work around the most reliable nodes, making it likely to be exceedingly resilient.

Clearly, the implementation of a distributed hash table would be advantageous. For the software distribution scenario some scheme of locating content is necessary, and as even BitTorrent, the most popular content sharing peer-to-peer application (Schulze & Mochalski, 2009) has officially adapted a DHT implementation for just purpose, it seems likely a DHT is an excellent solution and can possibly be used both for content and peer discovery.

Table 7 presents a comparison of the previously discussed DHT implementations. While none of the implementations explored would likely make for an unworkable solutions, it seems that Kademlia would be the most sound one. The especially enticing feature of Kademlia is its inherent ability to self-arrange to work with the most reliable nodes, possibly lending itself to alleviate the challenge of the circadian rhythm of nodes as well as the high mobility of some nodes. Also, it has been observed that piggybacking network stabilization on other network traffic is more efficient than explicit stabilization, being in some cases able to make it unnecessary, and parallel queries are better at lowering the lookup time during churn than faster stabilization (Li, et al., 2005). This is exactly what Kademlia does. Furthermore, a very compelling reason is its wide development in systems such as BitTorrent with a very large amount of users, from which it can be deducted with confidence that Kademlia is likely to scale and work well in the real world.

Table 7 - DHT comparison

	CAN	Chord	Pastry	Tapestry	Kademlia
Topology	d-dimensional Cartesian coordinate space	A unidirectional circular node ID space	A circular node ID space split into digits, route towards longest matching prefix taking locality into account	A circular node ID space split into words, route towards longest matching suffix	Binary tree organized around the most reliable nodes
Redundancy	Multiple independent coordinate spaces for each node	Store key-value pairs on multiple nodes with IDs succeeding the key	Replicate key-value pairs on multiple nodes with numerically closest IDs to the key	Incrementally salted keys stored on multiple nodes	key-value pairs stored on multiple nodes with closest matching IDs to the key
Availability	-	Several dated open source implementations exist	FreePastry, a dated BSD licensed Java implementation for R&D purposes	-	Maidsafe-DHT, BSD licensed open source C++ implementation, maintained and under development
Notable applications	-	CFS (Dabek, et al., 2001), OverCite (Stribling, et al., 2006), UsenetDHT (Sit, et al., 2008)	SCRIBE (Castro, et al., 2002), PAST (Druschel & Rowstron, 2001), SplitStream (Castro, et al., 2003)	-	BitTorrent (Loewenstern, 2008), Osiris Serverless Portal System (Kodeware, 2011)
Authentication and encryption	-	-	-	-	Maidsafe-DHT boasts an ability to "create digitally signed and secured p2p networks"

Trying to somehow take into account the physical structure of the network as opposed to simply routing by the overlay topology seems beneficial. The tapestry scheme of rebalancing based on the locality metric would probably not be very good for this application as the likely use case would indeed be for a network in which many hosts change location constantly, precisely what Tapestry proclaims not to be very efficient at supporting. The Pastry scheme might work well, but perhaps the best approach would be to introduce some additional local hierarchy such as one based on subnetting. If the network could be based on a peer-to-peer network of local cells with a supernode in charge, that supernode could then be taken advantage by using it to route data between cells and packing data destined to the master node as is the requirement, the advantages of which have been discussed in section 2.3.3.

While Skype boasts impressive security claims and tracker authentication and connection obfuscation can be implemented on top of the BitTorrent protocol, the other technologies reviewed offer no solution for the authentication problem, i.e. how to make sure a client is joining precisely the network it is supposed to, how to make sure only authorized clients can join the network, and how to make sure nobody outside the network can listen in on the communications. As it is, no published results exist on anyone being able to break the encryption or authentication of Skype and thus it seems likely that a public key scheme for authentication and symmetric key exchange would be a good option for security while using AES for symmetric encryption.

Based on these findings, it seems likely that a combined solution of a Skypelike hierarchical network combined with a reliable DHT would be a good solution for the previously presented challenges. The hierarchical side could be leveraged for NAT traversal and sound security while the DHT could provide true peer-to-peer data location. Kademlia seems like the best candidate as it appears to be not only theoretically suitable for the purpose but it has also found use in in massive real world applications as well as having a freely available and maintained implementation available.

4. The initial specification for the new communications architecture

This section discusses the specification first for the final product focusing on the client and then the prototype, presenting which features will be implemented in it. For both, the functionalities and the relevant operations are presented and the technologies used to implement those operations are specified. The basic protocol messages are also specified.

As stated in the first section, this specification is initial and is subject to change as the development process moves beyond the scope of this thesis. Further development is especially likely to happen for the protocol messages. As this is the case, this section is most detailed about things related to prototype and the structure and topology of the network while remaining on a higher level regarding things which are more of an application for the network to be developed later in the process.

First, the network structure and topology are discussed on a high level, after which the discussion moves on to encompass the routing schemes operating in the network. After these, the operations implementing the functionality are discussed. NAT traversal is then presented after which the section ends with the summarization of the satisfaction of the requirements presented previously in section 2.5. and a brief description of the prototype.

4.1. Network structure and topology

The network topology, as presented in Figure 8 combines elements from decentralized and hierarchical systems, and shall be composed of a hybrid of layers and a DHT. The basic component of the network is a node with special cases of Local Super Nodes (LSN) and Global Super Nodes (GSN), and a Master Node. This topology is chosen to facilitate the local buffering of information to achieve less bandwidth required to transmit low priority information such as device inventories, to facilitate the NAT traversal, to enable authentication and to make data location and routing in the overlay relatively simple affairs.

The node is the basic building block of the system. Once a client is distributed to a host, it becomes a node in the network by initiating the join operation. All hosts are eligible to become LSNs and all nodes which have a reachable address in the public internet may also serve as GSNs. Each subnet with nodes must have a designated LSN but the use of GSNs is more transient.

An LSN is a node the responsibility of which is to act as a buffer for low priority information, receiving the information from its subordinate nodes and periodically updating the master node with this information. In the first prototype, all of the LSNs will keep a direct connection open to the master node, but at a later phase it will be studied whether utilizing GSNs for relays or could be used for additional benefits.

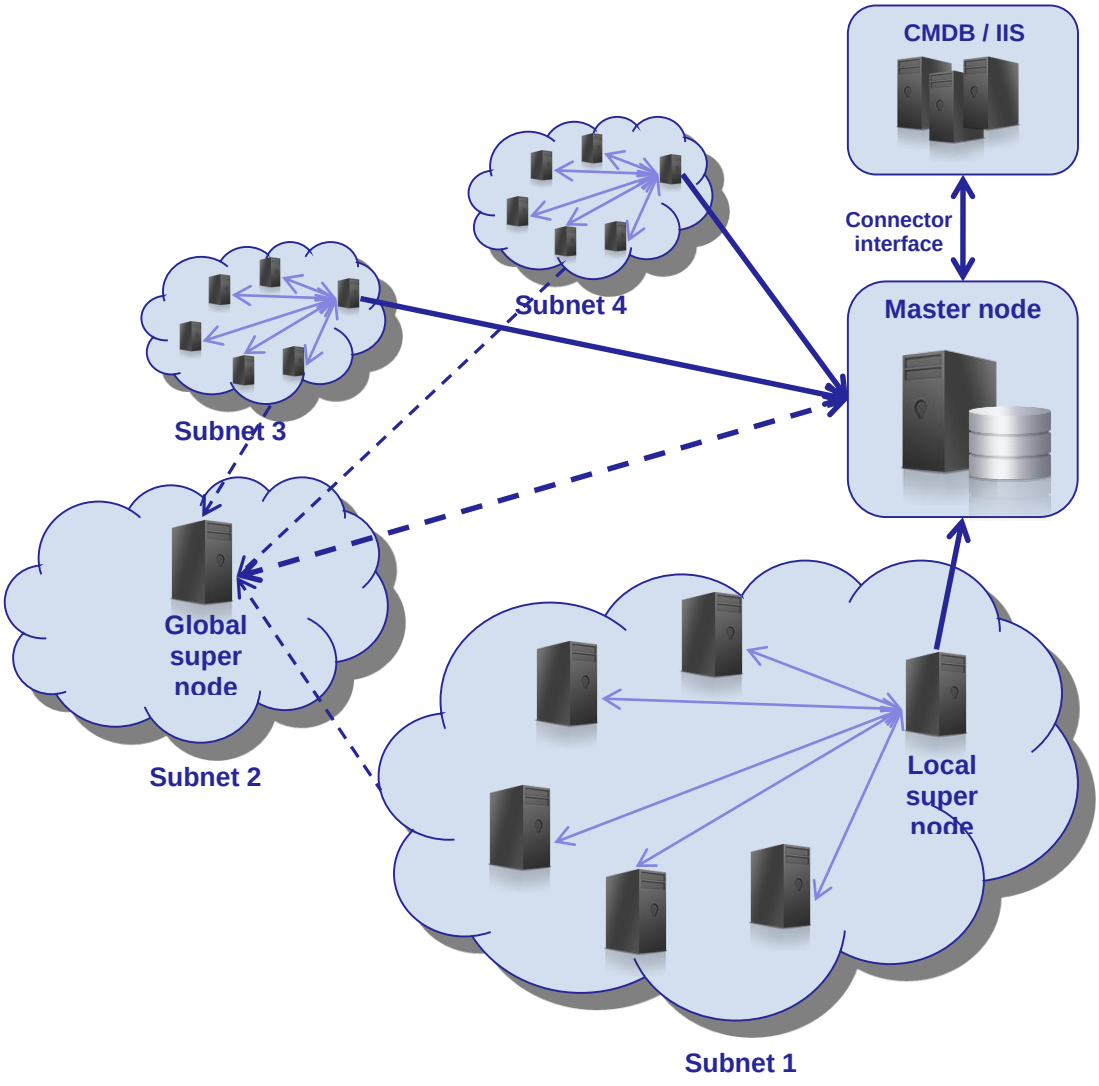

Figure 8 - the topology of the hierarchical network

A GSN is a Node which is responsible for facilitating NAT traversal. To be eligible for becoming a GSN, a node must have a public IP in the internet and have a GSN eligibility setting enabled. This setting exists so unwanted devices can be prevented from becoming GSNs. This would be sensible for example in the case of a highly mobile device which is known to have rapidly changing network reachability and possibly the additional desire to conserve as much resources as possible. Eligibility for GSN status is determined by having the master node and/or several GSNs contact a node with an inbound package and if the prospective node is able to receive these, it is deemed eligible. The contact information of GSNs can be stored in the DHT with reliability information so the most reliable ones may become the most frequently used. Also, LSNs can probe GSNs for some distance metric values such as network hops or round trip time periodically and disseminate this information to their subordinate nodes in order to take advantage of locality properties.

In addition to the hierarchical structure, an entirely decentralized Kademlia DHT is implemented to work between the nodes. It will be used to store the contact information of nodes and can be utilized also for the distributed installation point scenario for storing information on which nodes have which files. The possibility for using a free implementation Maidsafe-DHT (MaidSafe, 2012) for as much of the functionality as possible will be explored in the prototyping phase.

As illustrated in Figure 9, the topology of the Kademlia network can be represented by a binary tree. Assuming a single node is at the spot marked by the blue ball, it must and will have contact with the subtrees marked with the rectangles, with each rectangle representing the range of a k-bucket. As all hosts follow this pattern and as routing information, any host will be able to contact any host within a logarithmic number of hops in proportion to the number of nodes in the network. The parameter *k* defining the number of nodes in a k-bucket and the parameter α defining the number of parallel lookups when performing node or key queries will have to be experimented with during prototyping as will the expiration and republication intervals of key-value pairs and the k-bucket refreshing interval. The technology is discussed in more detail in section 3.1.5.

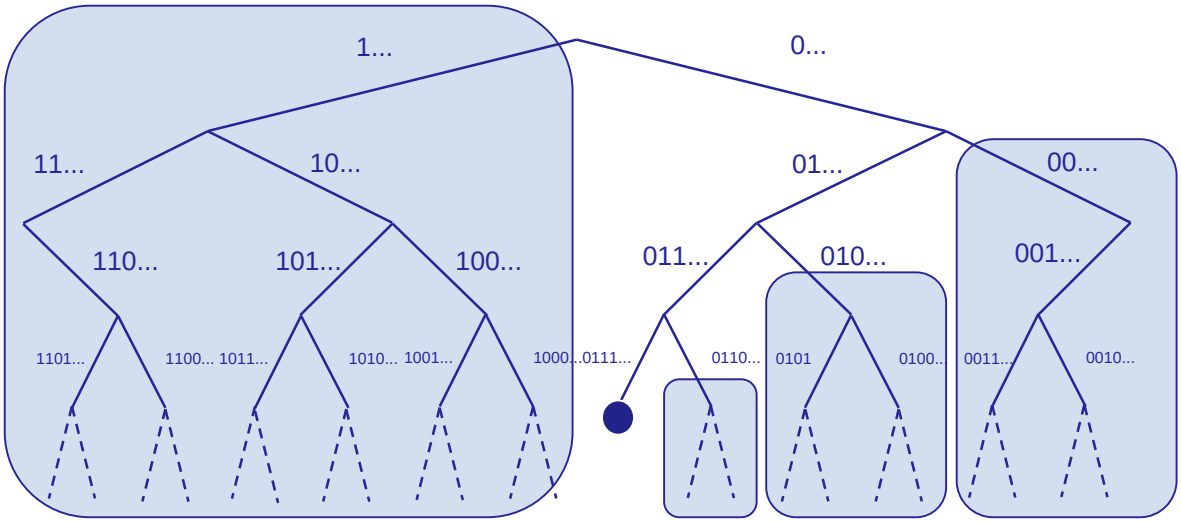

Figure 9 – Kademlia topology

As the routing scheme of Kademlia preferring the most reliable nodes, it may present an opportunity to exploit this information for GSN selection and possibly other purposes requiring reliable nodes.

4.2. Routing

Two routing schemes shall operate in parallel in the network. As the main function of the network is to route data to the master node, the route shall always be node -> LSN -> master node as presented in Figure 8. As a DHT will greatly facilitate using the network for arbitrary data storage, such as storing file-node mappings for installation media searching, the Kademlia protocol will also be implemented.

Each host shall be tagged with a 32-byte (160 bit) GUID which will also signify its address in the overlay address space. The GUID will be an SHA-1 hash generated randomly when the client is first installed. Arbitrary message routing in the network will utilize the Kademlia routing protocol as discussed in sections 3.5.1 and 4.1 with a separate node -> LSN -> master node logic working in parallel as discussed in section 4.1.

4.3. Operations

This section describes the operations used to build and maintain the network.

4.3.1. Joining the network

Before joining the network, a host must first know of a node already in the network. It is assumed that the address of the master node is always known, but the search is started by querying the local subnet with ping messages to determine whether any hosts are found. If so, the host connects itself to the LSN and joins the Kademlia overlay as well, otherwise the node declares itself an LSN and uses either some host it has previously been in contact with to join the Kademlia overlay or as a last resort, the master node.

4.3.2. Parting the network

A node can part the network either by announcing it with the part message or by simply disconnecting. As a part message is received, the receiving node simply cuts the connection and assumes the host is now down for the specified reason. No acknowledgement is sent back to the parting node due to it being likely that the parting reason is either the host or client program shutting down or some related reason. This means the node might not be able to receive it anyway and making it likely it couldn’t resend the part message even in case of no acknowledgement received.

A node is detected missing from the network when either an LSN notices it is no longer receiving periodical pings from it when it is detected unreachable by Kademlia. When this happens, the parting is handled just like for a node having sent a part message, simply logging the reason as lost connection.

In either case, the parting host is simply removed from the child table of the LSN it disconnected from.

4.3.3. Maintaining the health of the network

In addition to the k-buckets discussed in section 3.1.5; each node keeps a neighbor list maintained by their LSN. The neighbor list is arranged by the LSN in a descending order by the recent average uptimes of the nodes, and if the LSN goes down, the new LSN is the topmost node in the list. In case of a contest, the LSN is chosen randomly and the other LSNs become its child nodes. This should lead to a network which arranges itself around the most reliable nodes.

4.3.4. Storing and locating data

Locating data, in this case the contact information of nodes and the nodes storing certain files, is done through the implementation of the Kademlia DHT. The hashes of the files are used for keys for the files and the corresponding value is a list of client GUIDs who have that file. The Kademlia technology is discussed in more detail in section 3.1.5.

Some file transfer protocol will likely be needed to be implemented on top of the communications layer, a likely candidate being the striping file transfer mechanism of BitTorrent. However, the specification of such protocol is beyond the scope of this thesis.

4.4. Authentication and Encryption

While more of an application and not a property of the network itself, authentication and encryption are paramount in securing it. The details of the implementation are beyond the scope of this thesis, but applicable high level concepts of securing the network are presented.

A public key authentication and encryption scheme can work between the nodes. The master node can have a globally known public key that can be queried from the network. The master node can then be authenticated by a node verifying its public key vice versa. The trust

information is propagated by storing the public key and the GUID of the admitted node in the DHT. Nodes can then verify whether any node belongs to the network by querying the DHT or failing that, the master node. This authentication scheme is presented in Figure 10.

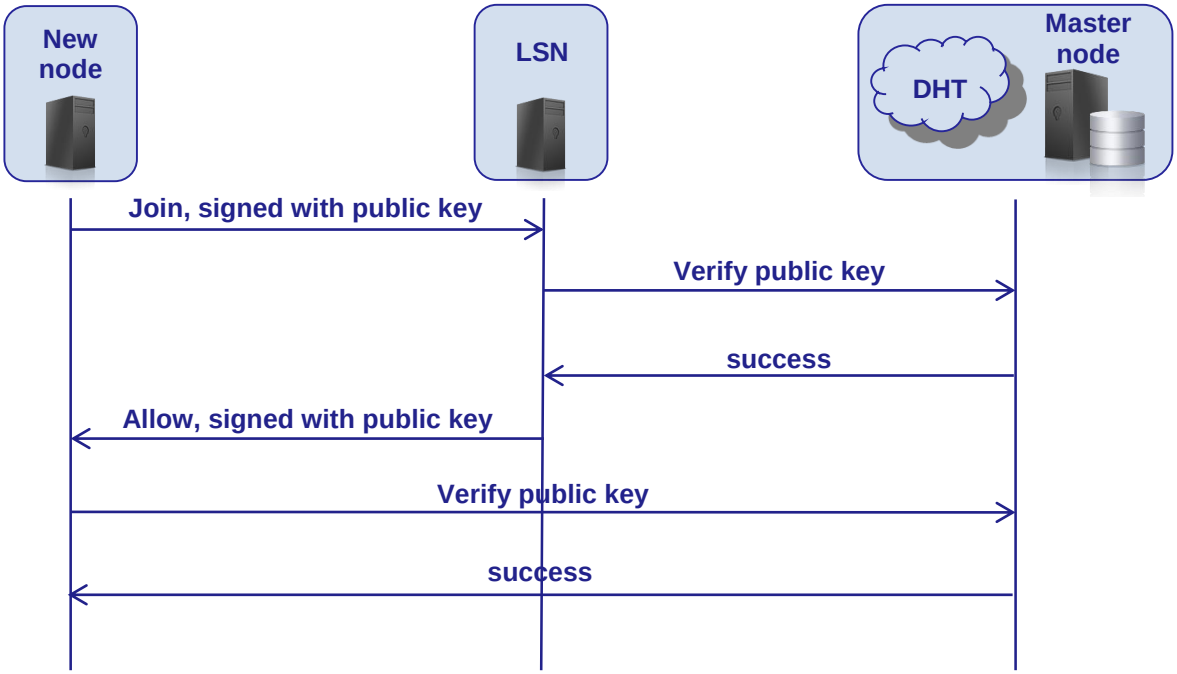

Figure 10 – Public key based authentication scheme 1

Another possibility is to have an one-time enrollment link for each installed client so the client can request an identity certificate using for example the Simple Certificate Enrollment Protocol (SCEP) (Pritikin, et al., 2011). The root certificate of these identity certificates is then published in the DHT and when nodes join, it is verified their identity certificate belongs to this trusted hierarchy. Network verification, i.e. a host joining the network verifying it is indeed the network it wants to join, is simple in the latter case in which it only has to verify the remote node's certificate belongs to a trusted hierarchy. In the former case, either the master node has to be used for verification, or if cached information about trusted GSNs exist, contact them for verification. This scheme is presented in Figure 11.

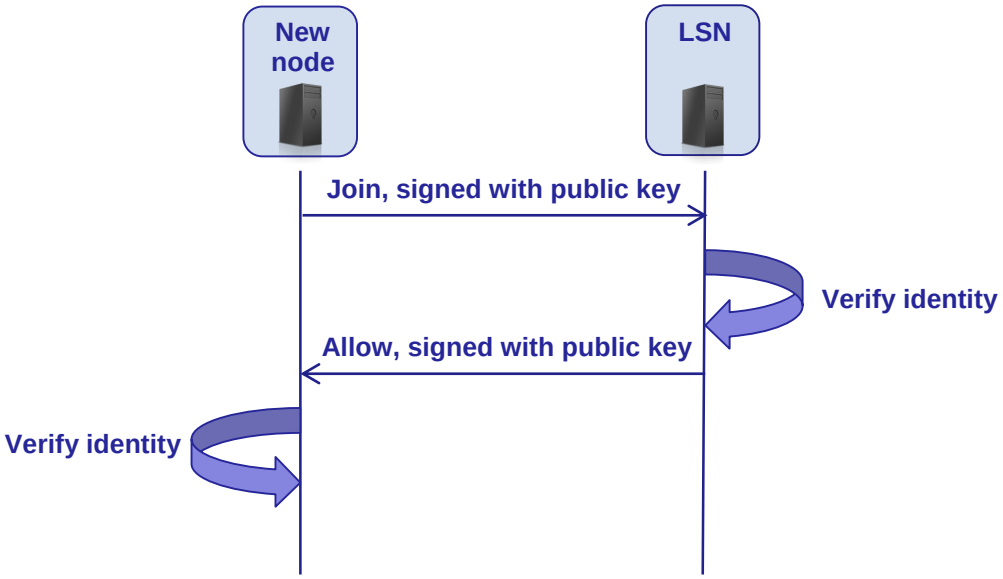

Figure 11 - Public key authentication scheme 2

For encryption, a public key scheme can also be used with nodes exchanging public keys prior to communicating with each other. Further study is needed on this and on whether for example Transport Layer Security (TLS) (Dierks & Rescorla, 2008) or the Message Stream Encryption protocol (Azureus Software Inc., 2012) supported by many BitTorrent clients could be used in the context of larger file transfers, or whether encryption is indeed needed for this purpose at all.

4.5. NAT Traversal

NAT traversal not being a part of the network per se, it is still very useful for the distributed installation point application. Without NAT traversal, many potential hosts could be left out of the file transfer swarms.

As presented in Figure 12, NAT traversal can be achieved through several ways. A super node can be utilized to facilitate UDP hole punching (Ford, et al., 2005) to allow two-way connection between the hosts wishing to communicate with each other, or arbitrary messages can be directly routed between hosts using the sendto message. The latter, however is more likely to cause congestion as there are no guarantees the routing node is in a sensible position

in the network topology or that the connection is fast enough. Thus, it is not recommended to use the sendto message for transferring large amounts of data. Additionally, if only one of the nodes is NATted, the super node may be used to facilitate connection reversal.

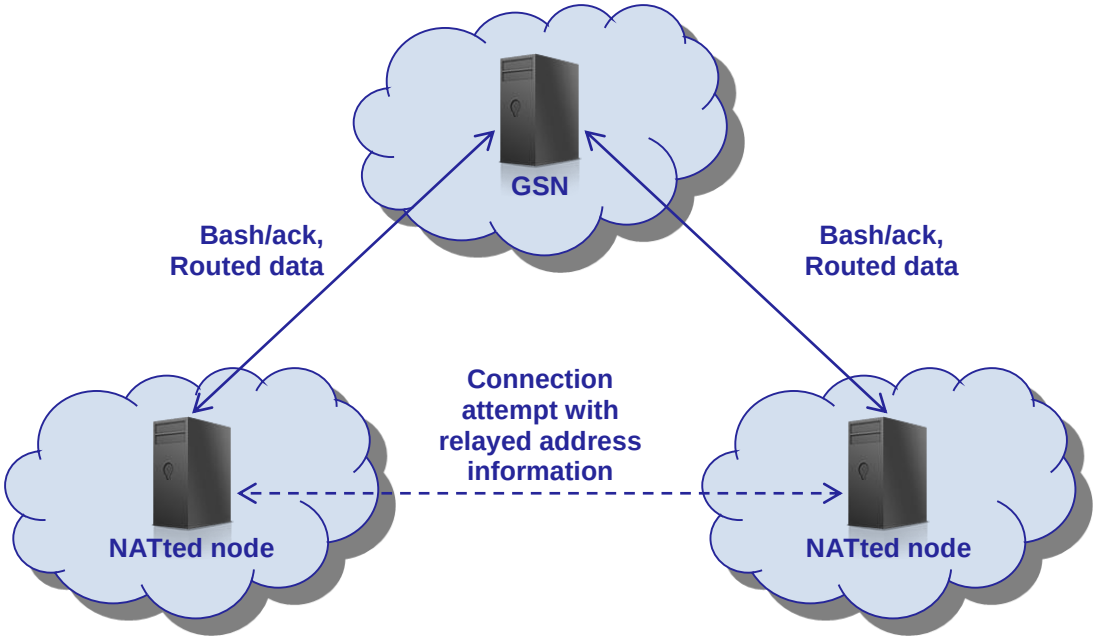

Figure 12 - NAT traversal

UDP hole punching works by two hosts contacting a third party acting as a mediator. During the initial contact, the address translators in between allocate temporary addresses for the hosts. The mediating party then relays the temporary address of each host to the other one, allowing direct connection. In the case of this solution, a super node will act as the mediating party between any two nodes.

4.6. Summary of the satisfaction of the requirements

This section takes the requirements as specified in section 2.5.1. and presents the summarized solutions for each one by one.

Table 8 - Requirement satisfactions of the topology and the routing

Requirement	Method of satisfaction
1	As it is possible for each node to directly contact the master node, there will be no impeding of direct communication between the nodes and the master node, enabling all current functionality to keep doing so with minimal modification. Furthermore, the new system can be built to work in parallel with the existing communication framework in order to maintain backwards compatibility for a transitional period.
2	Both of the topologies and their respective routing schemes presented in section 4.1. and 4.2. are known to be scalable. Skype with a similar node-supernode approach has seen over 40 million concurrent users (Caukin, 2012) and Gnutella with its peer-ultrapeer approach over 3 million (Stutzbach, et al., 2007). Also, it is known that the Kademlia based Azureus DHT has functioned well with over a million daily users (Falkner, et al., 2007).
3	While more of a property of the client program itself and not so much of the communications model, the LSNs do lend themselves to the purpose as they can rapidly detect any node leaving their subnet and facilitate quick joining when a node enters a subnet the LSN of which it remembers. Additionally, even when entering a new subnet, as long as internet access is available, a connection can be maintained to a GSN or to the master node in order to be connected to the client swarm as much of the time as possible.
4	As a DHT, Kademlia is very suitable for the foundation of a distributed network storage system. It can be utilized either directly for storing arbitrary key-value pairs or using the key-value storage function to for example store the contact information of hosts containing a certain file.
5	Bandwidth management is made possible by the LSN topology of the subnets. An LSN can be built to track the bandwidth usage of its subordinate nodes which in turn can be built to limit their bandwidth usage as instructed by the LSN. Measurements could be made to estimate

	the bandwidth available between the subnet and an endpoint and the bandwidth limitation adjusted accordingly.
6	As the LSNs are picked on a criteria based on the likelihood of a node remaining up in a given subnet and as Kademlia also organizes itself around the most reliable nodes, it can be argued that the architecture is very resilient against churn. Also, even in the case of losing contact with all other nodes, so long as an internet connection is available, it is always possible to connect directly to the master node.
7	Purely a feature of the client, possible to implement.
8	While some network requirements cannot be circumvented, such as the necessity for the client program to be able to connect to the internet and to local nodes, the topology which has the LSNs keeping in touch with the master node, the necessity to open up a listening port for clients to be woken up is obsoleted. Instead, any client can be contacted through its designated super node.
9	As the new communications paradigm no longer has the clients polling the server but two-way communication, any node available in the network can be contacted at any time, allowing the sending of instant messages.
10	Arbitrary host history information can be gathered and stored on the hosts themselves. As the communications paradigm allows two-way communications, the master node can query this information from an available node at any given time.
11	Encryption can be built into the network communications, both between peers and if need be, separately for end-to-end purposes so even if the messages are routed through other nodes, only the end nodes can decrypt the communications.
12	Host authentication can be supported. Each node can have an identity certificate and all messages signed with it. The certificate can then be verified to either be explicitly trusted or as belonging to a trusted hierarchy. Certificate information can be requested from the master node or stored in the network using the DHT.

13	The identity certificate and the public key of the master node can be distributed with each client and the client can then always authenticate any message from the master node. Additionally, if this is the root certificate of all of the clients' identity certificates, the clients can verify any client as belonging to this hierarchy.
14	NAT traversal can be supported as defined in section 4.5. Additionally, if the NAT traversal features of Maidsafe-DHT can be used, they may provide an efficient way of implementation.
15	IPv6 support is purely a feature of the client and can be implemented.

4.7. The Prototype

The purpose of the prototype is to serve as a proof-of-concept for the network topology. To achieve this, it will implement the operations ping, pong and update as well as study the possibility for using maidsafe-dht for the DHT implementation. The primary feature will be the Node - Local Super Node - Master Node topology with presence information routed to the master node so the master node always knows the nodes present in the network and the super node through which they are connected. The implementation and the methods of satisfying the requirements as specified in section 2.5.2. and 2.5.3. will be discussed in detail in sections 5 and 6.

5. Implementation

This section discusses the implementation of the prototype. The phases of the work are presented one by one and for each the methods of implementation, encountered problems and achievements are reported.

5.1. Used technologies

A preliminary study was done to determine the technologies used in the development of the prototype. The BSD-Licensed Kademlia implementation Maidsafe-DHT was explored as an implementation option but was quickly deemed too uncertain an option for the proof-of-concept purpose. While the code itself appeared to work and the demo programs successfully built a functional Kademlia network, the documentation was nonexistent and the code exceedingly lacking in comments. Thus, after some fruitless research, it was determined that while providing the opportunity of possibly being able to use the C++ code from this prototype option for the final product, the cost of learning the technology was deemed too much of an unknown taking into account the scope of the project and the priority of producing a proof-of-concept as opposed to a real customer product.

As the prototype was also only required to work in the Microsoft Windows environment, the .NET framework and the C# programming language were selected as the tools for implementation. The prototype was implemented from scratch using these tools and the Microsoft Visual Studio 2012 development environment.

5.2. High level description of the communications model

The prototype network implements the previously discussed Node – Local Super Node – Master Node using the UDP protocol and port 31337 for communications. The Nodes keep the LSNs updated about their joins and parts and the LSNs poll the Nodes periodically to detect Nodes which were lost but did not send a part message. The LSNs in turn keep the

Master Node updated of the status of their subnets whenever changes take place. Like the LSNs poll their subordinate Nodes, the Master Node polls the LSNs to detect when and if they go down unannounced. Figure 13 presents the high level operation structure of the prototype network.

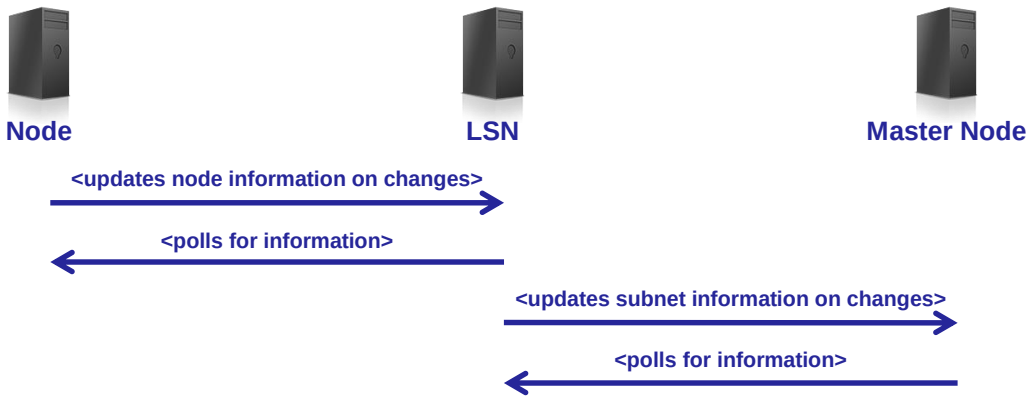

Figure 13 - The high level functions of the prototype network components

5.3. Features of the node

The prototype is a proof-of-concept for the Node-LSN-Master node topology of the network as well as simulating a DHT for the simple application of storing the super node information of the network. Nodes can join and part from the network with the master node keeping track of the network topology at any given time. The availability of nodes is tracked through the master node periodically sending queries to them eliciting a response and measuring the round trip time and detecting dropped nodes.

The program is essentially a state machine with five possible states in addition to the special case of being the master node, as depicted in Figure 14. The following subsections discuss these states, the functions of each and the transitions in more detail.

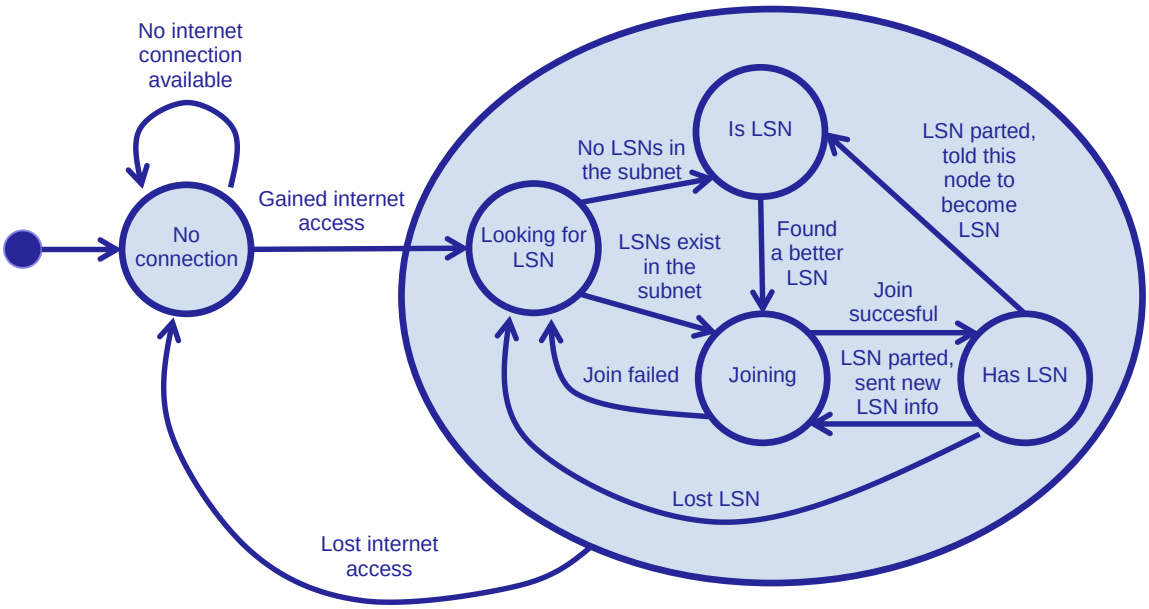

Figure 14 - the prototype state machine

5.3.1. No connection

The program starts up in the no connection state. It then periodically polls through all the up network interfaces with IPv4 addresses associated with them for one connected to the internet, and finding one, binds a UDP socket onto that IP port 31337 for listening and sending of data and enters the Looking for LSN state.

5.3.2. Looking for LSN

In the Looking for LSN state, the program sends an LSNQuery message to the master node. An empty ack as a response signifies no other LSNs exist in the subnet and the new node becomes an LSN for that subnet as illustrated in Figure 15. The new LSN then sends a join message to the master node which acknowledges the join with an Ack.

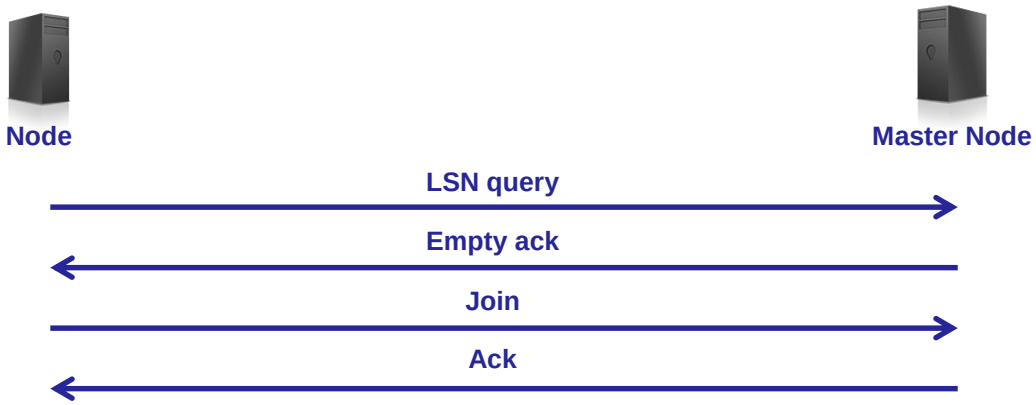

Figure 15 - Node joining as an LSN

If the master node responds with a node list, the node parses the nodes from the list marked as LSNs and enters the Joining state as illustrated in Figure 16.

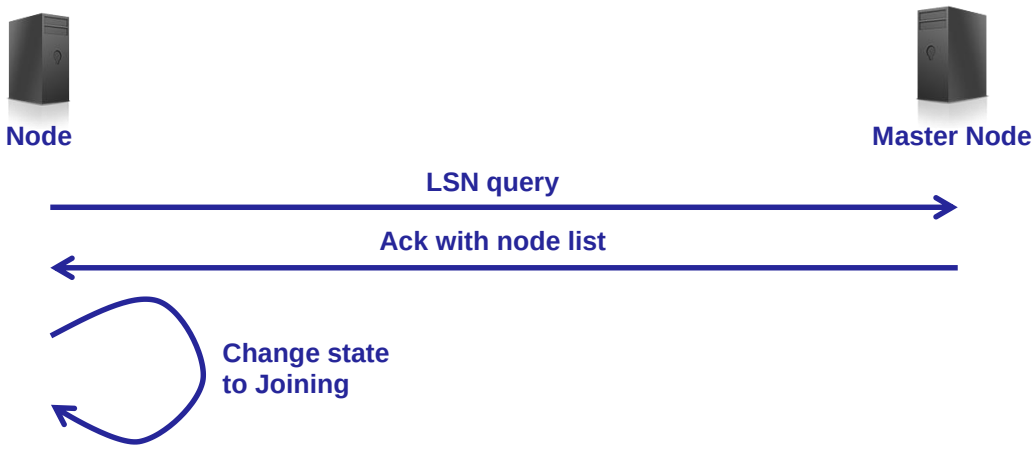

Figure 16 - A new node in a subnet gets a node list

5.3.3. Joining

In the Joining state, the node periodically sends Join messages to each of the LSNs it parsed from the response to the LSNQuery received from the master node. If it receives an Ack message to a Join message, it transitions to the Has LSN state as illustrated in Figure 17. If no LSN in the list responds, the node reverts back to the Looking for LSN state.

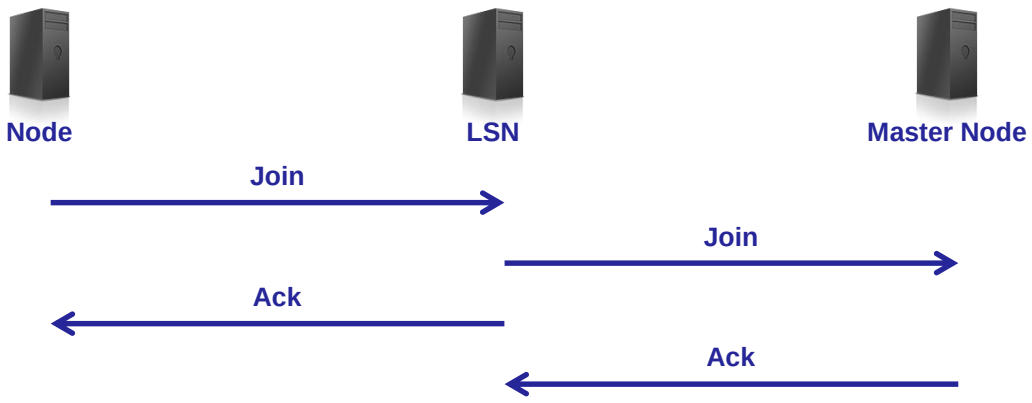

Figure 17 - A new node joins an LSN

5.3.4. Has LSN

The Has LSN state is the main functional state of a basic node. The primary function of the node in this state is to respond to any messages sent by its LSN. In case nothing is received, the node also periodically polls the LSN with Ping messages to check that it's still alive. In most cases, however, the connection should be kept alive by the LSN polling the subordinate. All messages to and from the Master node apart from direct debug messages, are now routed through the LSN. Messages from nodes other than the LSN will be dropped, except for Ping messages, to which a Pong message containing the information of its LSN. This is done so in the case of some other node thinking this node is an LSN will notice it is not and connect to the proper LSN.

5.3.5. Is LSN

In the Is LSN state, the node functions as the super node of its subnet. It routes all traffic between its subordinate nodes and the master node as well as keeps track on whether its subordinate nodes are still alive by periodically sending Ping messages as illustrated in Figure 18.

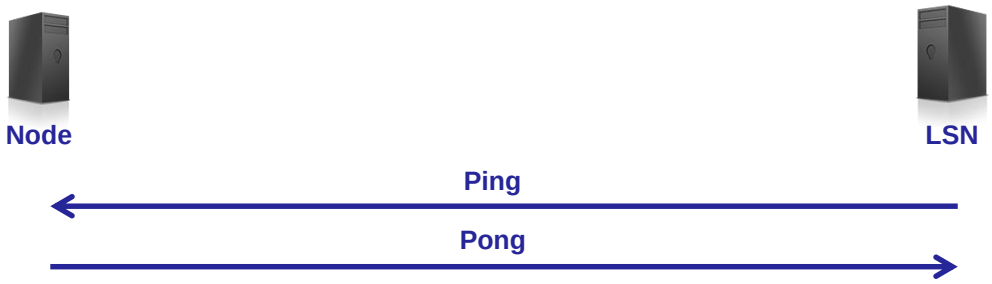

Figure 18 - The ping-pong message-response pattern of LSNs and nodes

The LSN also periodically sends LSNQuery messages to the MN and if another LSN is detected in the subnet, sends it an Uptime message containing its uptime. If the receiving LSN has a larger uptime, it responds with its own Uptime message. If the uptime of the recipient LSN is lower, it sends Part messages containing the information of the better LSN to all its subordinates and sets its state to Joining. The subordinates will also set their state to Joining and try to join the better LSN. This behavior is illustrated in Figure 19.

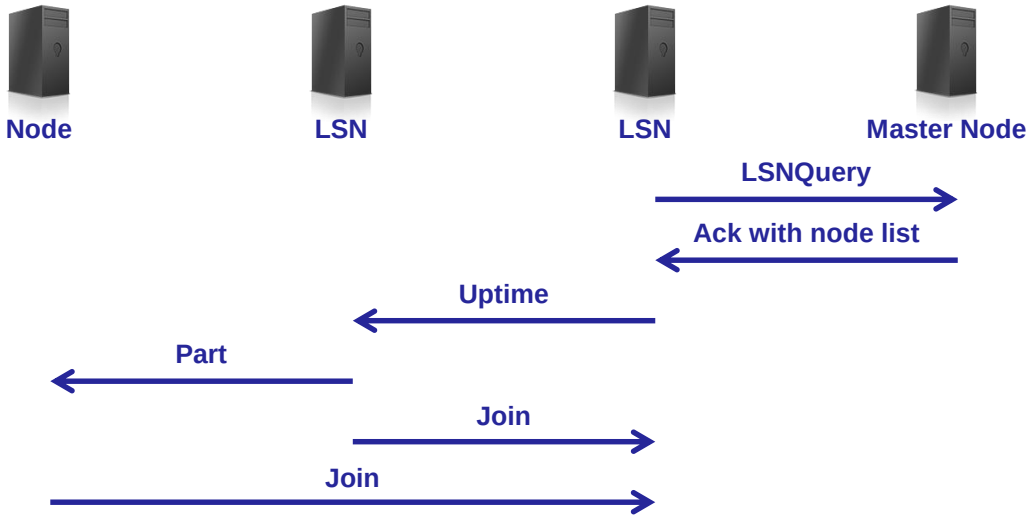

Figure 19 - Competing LSNs solve their priorities

In case some of these messages are lost, the nodes will detect the change in LSN the next time they attempt to poll the former LSN, as it will respond to their Pings with Pongs containing the address of the new LSN.

5.4. Features of the master node

The master node acts as the central tracker of the network for the purposes of the prototype. It keeps track of the LSNs and their subordinates, sends Ping messages to the LSNs as well as the subordinates to keep track of the network status. The information is stored in a hash table. The key is an SHA-1 hash of the local network address of the node concatenated with the public network address of the node assuming the same network mask with the local address. The values contained in the hash map are the GUID, local address and state of each node in the network, as well as a timestamp they were last seen.

5.5. Measurements

This section presents the measurements produced by the prototype test environment. The test environment comprised of the desktop and laptop computers, servers and virtual machines of Miradore R&D personnel. Test computers were recruited by e-mailing the entire R&D personnel asking everyone to install the prototype onto as many machines as possible and requesting log files after a period of time. While the total number of running nodes remained unknown, it is assumed that the log files recovered represented either 100% or very close of the entire test environment.

As the absolute priorities were the ease of configuration and reliability of the real time connection in any conditions, the emphasis on the results is on these. Also, as scalability is an important issue and used bandwidth is a point of interest, these are also explored. Causes of unreliability and connection problems are explored on a case by case basis while bandwidth measurements were taken from a selected LSN.

It was determined that the circadian rhythm was indeed present with each 24 hour period yielding very similar results. Thus, the measurement period was selected as 24 hours. The date the measurements were taken was Thursday 2012-12-20 between 00 and 24 hours.

5.5.1. Measurements at the nodes.

The main measurement at the nodes is concentrated on reliability. Every time a connection to the network is lost or gained to the network, the nodes logged the reason of the disconnection and amount of time it took to regain a connection. During the measurement period there were nine active nodes in addition to the LSN in total, from which the measurements presented in Table 9.

Table 9 - node recovery measurements

Mean	Median	Minimum	Maximum	Standard deviation
2.9s	2.6s	2.5s	5.3s	0.9s

The most common encountered problem case at the nodes was that some firewall programs prevented the use of UDP, sometimes symmetrically and sometimes asymmetrically. While the asymmetric case should not be a problem when handled correctly, it did on occasion lead to a situation where an LSN could emerge in a subnet to which no other node could connect to. This situation was solved by creating the hash table feature on the MN, allowing for nodes to question the MN about the LSNs in any network and attempting to connect to any one of those. Eventually the LSNs would also discover each other through LSN update messages and manage to solve the issue, but considerable overhead in communications and response times could be incurred.

The other problem encountered was similar but not identical – the complete lack of UDP connectivity. The prototype tests internet connectivity through attempting TCP connections to several HTTP servers in the internet, which proved much more reliable than the custom UDP protocol used in the communication between the nodes.

5.5.2. Measurements at the LSN

As predicted in section 4.3.3; it turned out the network arranged itself readily around a node which was very reliable, in this case always on. In the following measurements the scalability is explored through measuring used LSN-MN bandwidth in relation to the number of subordinate nodes of the LSN.

The measurements were somewhat complicated by the network being infested by buggy old versions of nodes which created some noise by keeping a constant background noise of part messages in the network. Additionally, a buffering bug resulted in considerable overhead in the protocol. However, as the program is a prototype, problems were expected and knowing the issues, it was possible to filter the results so they represent the state of a properly functioning network with no payload traffic. Figure 20 shows bandwidth used by sent message type from LSN to MN, showing that most of the activity in the network comes from the rather uneconomical use of the hash table at the master node with updates of the subnet state always updating the LSN information of the entire subnet instead of just the changes.

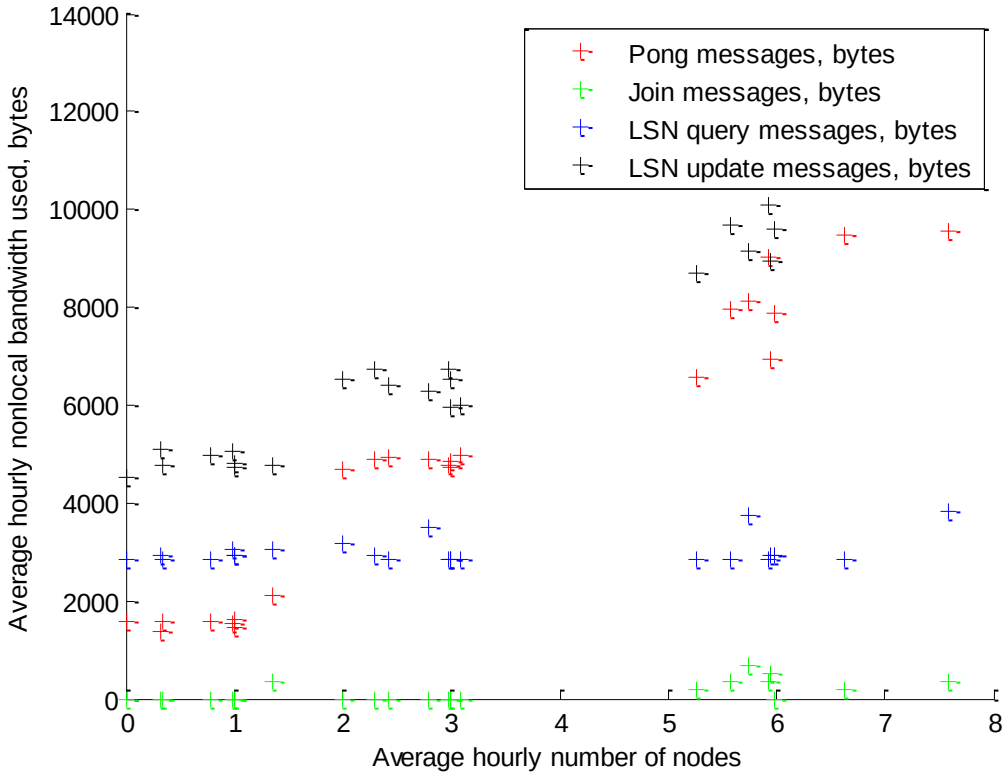

Figure 20 – bandwidth used by sent message type from LSN to MN

Figure 21 also includes the ack messages received in reply to the update and query messages, further illustrating the point.

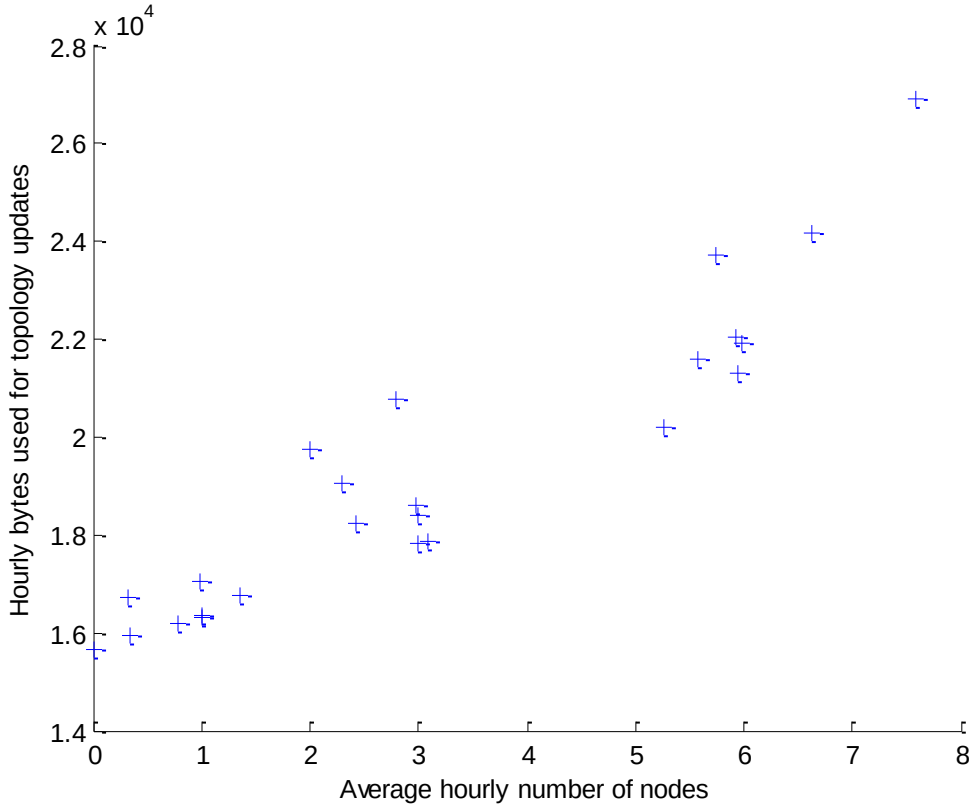

Figure 21 - bandwidth used for topology update messages between LSN and MN

Figure 22 presents the nonlocal bandwidth used per a local node ignoring the prototype overhead, i.e. this only includes the join messages and the keepalive messages used to maintain the connection between the LSN and the MN. While due to a buffering bug part messages are not a part of this plot, from this it could still with relatively high confidence be extrapolated that if the behavior of the DHT to be developed can be optimized such that it is only updated when changes occur, relatively little bandwidth is required per node, clearly in the order of a couple of kilobytes per hour.

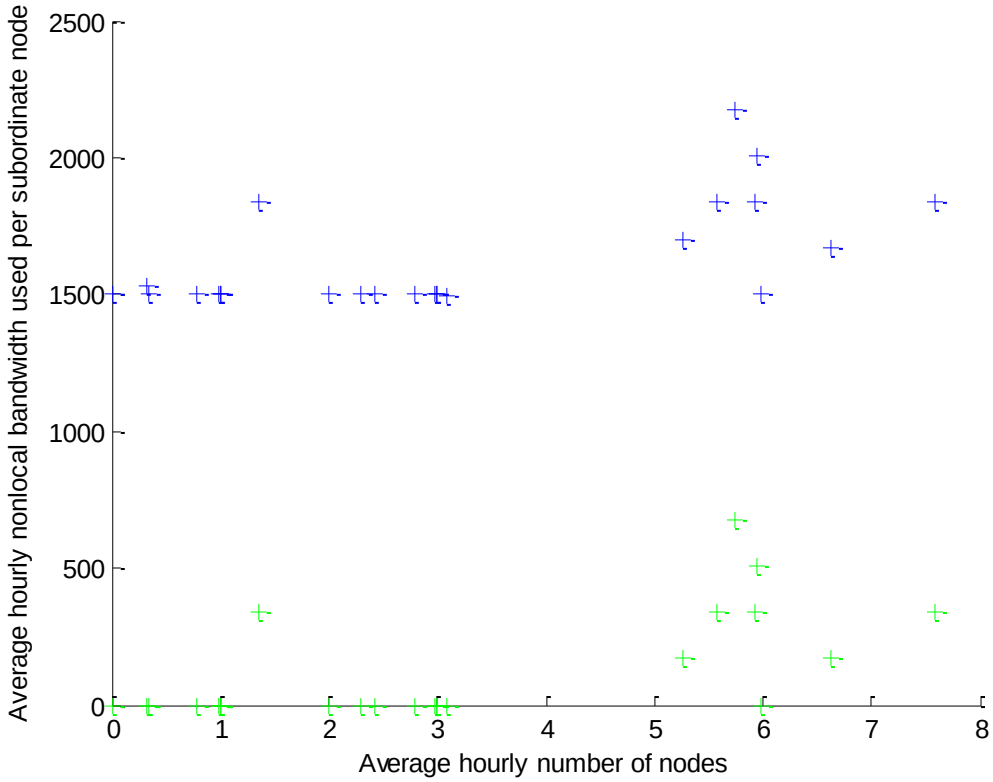

Figure 22 - overhead corrected used bandwidth per node

6. Conclusion

The research, the resulted specification and the prototype together demonstrate that the new model for communications is feasible and clearly provides a way to implement the requirements as presented in section 2.5.1. While the prototype is uneconomical in its use of bandwidth, a great achievement is that any node gaining a network connection is connected to the network in under three seconds on average. In combination with this, the ability for the MN to contact any node currently connected to the network at any given time and all this working simply by installing the prototype program, it can be said that the most important requirements were clearly met.

While the specification and the prototype do in combination serve as a small scale proof of concept, there is still clearly much development and productization to do before the system is ready for deployment in a production environment. Further prototyping, development and exploration will be required into the details of implementation such as encryption and authentication as well as much optimization needed for the use of bandwidth. Furthermore, the prototype does not at all implement the GSN layer or NAT traversal, both of which will need to be implemented and tested.

The used protocol choice of UDP also proved somewhat problematic. Several users reported having to manually configure their firewalls to allow for in and outbound traffic on the used UDP port while the TCP based internet connection testing of the prototype worked without a hitch. This hints that TCP might be better suited as a protocol or at least it should be implemented as a backup option to increase reliability. Another problem case was the asymmetry which happened when a node could establish an outbound UDP connection but could not receive any. This could cause a situation in which a LSN could emerge to which no node could gain a connection.

Even though the prototype does prove the feasibility of the concept on a small scale, the amount of work needed to totally finalize the system and all its components is likely to be massive. Additionally, as the tests done were on a relatively small scale compared to the

prospective size of a real deployment, scalability may need further verification. However, based on the conducted research and the results, it can be said that the new model specified in this paper is workable. The next step in development is to estimate the amount of work needed to make the specification into a product, evaluate the importance of the project taking into account the resources needed and make further decisions based on that.

Bibliography

Abu Rajab, M., Zarfoss, J., Monrose, F. & Terzis, A., 2006. *A Multifaceted Approach to Understanding the Botnet Phenomenon.* Rio de Janeiro, ACM.

Azureus Software Inc., 2012. *Message Stream Encryption.* [Online] Available at: http://wiki.vuze.com/w/Message_Stream_Encryption [Accessed 19 09 2012].

Bailey, M., Cooke, E., Jahanian, F. & Xu, Y., 2009. *A Survey of Botnet Technologies and Defenses.* Washington DC, IEEE.

Barford, P. & Yegneswaran, V., 2007. An Inside Look at Botnets. *Advances in Information Security,* Volume 27, pp. 171-191.

Baset, S. A. & Schulzrinne, H., 2004. *An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol,* New York: Columbia University Department of Computer Science.

Caizzone, G., Corghi, A., Giacomazzi, P. & Nonnoi, M., 2008. *Analysis of the Scalability of the Overlay Skype System.* Beijing, IEEE.

Castro, M. et al., 2003. *SplitStream: high-bandwidth multicast in cooperative environments.* Bolton, ACM, pp. 298-313.

Castro, M., Druschel, P., Kermarrec, A.-M. & Rowstrom, A., 2002. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications,* 20(8), pp. 100-110.

Caukin, J., 2012. *The Big Blog: Everything new at Skype's HQ - 40 Million People: How Far We've Come.* [Online] Available at: http://blogs.skype.com/en/2012/04/40_million_people_how_far_weve.html [Accessed 10 07 2012].

Chawathe, Y. et al., 2003. *Making Gnutella-like P2P Systems Scalable.* Karlsruhe, ACM, pp. 407-418.

Chordless, 2011. *Chordless.* [Online] Available at: http://sourceforge.net/projects/chordless/ [Accessed 11 09 2012].

CNN Tech, 2000. *Open source Napsterlike product disappears after release.* [Online] Available at: http://articles.cnn.com/2000-03-15/tech/gnutella_1_open-source-open-beta-unauthorized-freelance-project [Accessed 14 07 2012].

CNN, 2002. *Napster files for bankruptcy.* [Online] Available at: http://money.cnn.com/2002/06/03/news/companies/napster_bankrupt/ [Accessed 12 07 2012].

Cohen, B., 2008. *The BitTorrent Protocol Specification.* [Online] Available at: http://www.bittorrent.org/beps/bep_0003.html [Accessed 12 07 2012].

Cooke, E. & Jahanian, F., 2005. *The Zombie Roundup: Understanding, Detecting, and Disrupting Botnets.* Cambridge, USENIX.

CURRENT Lab, U., 2006. *Chimera Downloads.* [Online] Available at: http://current.cs.ucsb.edu/projects/chimera/download.html [Accessed 14 09 2012].

Dabek, F. et al., 2001. *Wide-area cooperative storage with CFS.* Banff, ACM, pp. 202-215.

Dan-To Services, 2012. *Beacon Cache II 0.8.0.1.* [Online] Available at: http://beacon2.technutopia.com/gwc.php [Accessed 15 07 2012].

De Cicco, L., Mascolo, S. & Palmisano, V., 2007. *An Experimental Investigation of the Congestion Control Used by Skype VoIP,* Bari: Politecnico di Bari Dipartimento di Elettrotecnica ed Elettronica.

Desclaux, F. & Kortchinsky, K., 2006. *Vanilla Skype.* Montreal, REcon.

Dierks, T. & Rescorla, E., 2008. *The Transport Layer Security (TLS) Protocol,* s.l.: IETF.

Druschel, P. & Rowstron, A., 2001. *PAST: A Large-Scale Persistent Peer-to-Peer Storage Utility*. Elmau, s.n.

Dämpfling, H., 2003. *Gnutella Web Caching system.* [Online] Available at: http://www.gnucleus.com/gwebcache/ [Accessed 15 07 2012].

Falkner, J. et al., 2007. *Profiling a million user DHT.* San Diego, ACM, pp. 129-134.

Ford, B., Srisuresh, P. & Kegel, D., 2005. *Peer-to-Peer Communication Across Network Address Translators.* Anaheim, Usenix.

Fox, G., 2001. Peer-to-Peer Networks. *Computing in Science & Engineering*, 3(3), pp. 75-77.

Free Software Foundation, 2007. *GNU General Public License.* [Online] Available at: http://www.gnu.org/copyleft/gpl.html [Accessed 01 08 2012].

FreePastry, 2009. *Pastry - A scalable, decentralized, self-organizing and fault-tolerant substrate for peer-to-peer applications.* [Online] Available at: http://www.freepastry.org/ [Accessed 10 09 2012].

Goodin, D., 2012. *Ars Technica: Skype replaces P2P supernodes with Linux boxes hosted by Microsoft (updated).* [Online] Available at: http://arstechnica.com/business/2012/05/skype-replaces-p2p-supernodes-with-linux-boxes-hosted-by-microsoft/ [Accessed 10 07 2012].

Grossel, Y., 2000. *gtk-gnutella - The Graphical Unix Gnutella Client.* [Online] Available at: http://gtk-gnutella.sourceforge.net/ [Accessed 14 07 2012].

Hand, S. & Roscoe, T., 2002. *Mnemosyne: Peer-to-Peer Steganographic Storage.* Cambridge, MA, s.n.

Ilie, D., Popescu, A. & Nilsson, A. A., 2004. *Measurement and Analysis of Gnutella Signaling Traffic.* Stockholm, s.n.

Jaanus, 2006. *The Big Blog: Everything new at Skype's HQ - 5 million online Skypers.* [Online]
Available at: http://blogs.skype.com/en/2006/01/5_million_online_skypers.html
[Accessed 10 07 2012].

jDHTUQ, 2010. *jDHTUQ.* [Online]
Available at: http://sourceforge.net/projects/jdhtuq//
[Accessed 10 09 2012].

Klingberg, T. & Manfredi, R., 2002. *Gnutella Protocol Development.* [Online]
Available at: http://rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.html
[Accessed 16 07 2012].

Kodeware, 2011. *Osiris - Serverless Portal System.* [Online]
Available at: http://www.osiris-sps.org/
[Accessed 12 09 2012].

Liebowitz, S. J., 2006. File Sharing: Creative Destruction or Just Plain Destruction. *Journal of Law and Economics*, 49(1), pp. 1-28.

Li, J. et al., 2005. *A performance vs. cost framework for evaluating DHT design tradeoffs under churn.* Miami, IEEE, pp. 225-236.

Lime Wire LLC, 2000. *Official LimeWire Website - Lime Wire.* [Online]
Available at: http://www.limewire.com/
[Accessed 14 07 2012].

Link, K., 2012. *Beacon Cache II 0.8.0.1.* [Online]
Available at: http://beacon.numberzero.org/gwc.php
[Accessed 15 07 2012].

Loewenstern, A., 2008. *DHT Protocol.* [Online]
Available at: http://bittorrent.org/beps/bep_0005.html
[Accessed 18 07 2012].

MaidSafe, 2012. *maidsafe-dht - c++ DHT (kademlia) with NAT traversal and cryptographic libraries.* [Online]
Available at: http://code.google.com/p/maidsafe-dht/
[Accessed 01 08 2012].

Maymounkov, P. & Mazières, D., 2002. *Kademlia: A Peer-to-peer Information System Based on the XOR Metric.* Cambridge, MA, s.n.

Michelangeli, E. & Jalkanen, A., 2006. *KadC - P2P library.* [Online]
Available at: http://kadc.sourceforge.net/
[Accessed 01 08 2012].

Open Chord, 2011. *Open Chord.* [Online]
Available at: http://sourceforge.net/projects/open-chord/
[Accessed 11 09 2012].

Open Source Initiative, 2012. *Open Source Initiative OSI - The BSD License:Licensing.* [Online]
Available at: http://opensource.org/licenses/bsd-license.php
[Accessed 01 08 2012].

Perényi, M., Gefferth, A., Dinh Dang, T. & Molnár, S., 2007. *Skype Traffic Identification.* New Orleans, IEEE.

Portmann, M., Sookavatana, P., Ardon, S. & Seneviratne, A., 2001. *The Cost of Peer Discovery and Searching in the Gnutella Peer-to-peer File Sharing Protocol.* Bangkok, IEEE, pp. 263-268.

Pritikin, M., Nourse, A. & Vilhuber, J., 2011. *Simple Certificate Enrollment Protocol draft-nourse-scep-23,* s.l.: IETF.

Ratnasamy, S. et al., 2001. *A Scalable Content-Addressable Network.* San Diego, ACM.

Rhea, S. et al., 2003. *Pond: the OceanStore Prototype.* San Francisco, USENIX.

Ripeanu, M., Forste, I. & Iamnitchi, A., 2002. Mapping the Gnutella Network: Properties of Large-Scale Peer-to-Peer Systems and Implications for System Design. *IEEE Internet Computing Journal,* 6(1), pp. 50-57.

Rohrs, C., 2001. *Query Routing for the Gnutella Network.* [Online] Available at: http://rfc-gnutella.sourceforge.net/src/qrp.html [Accessed 18 07 2012].

Rowstron, A. & Druschel, P., 2001. *Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems.* Heidelberg, Middleware.

Sauer, K., 2007. *ZDnet / Security / Analysen: Telefonieren übers Internet: Wie sicher ist Skype wirklich?* [Interview] (13 February 2007).

Schulze, H. & Mochalski, K., 2007. *Internet Study 2007,* s.l.: Ipoque.

Schulze, H. & Mochalski, K., 2009. *Internet Study 2008/2009,* s.l.: Ipoque.

Singla, A. & Rohrs, C., 2001. *Ultrapeers: Another Step Towards Gnutella Scalability,* s.l.: Lime Wire LLC.

Sit, E., Morris, R. & Kaashoek, F. M., 2008. *UsenetDHT: a low-overhead design for Usenet.* San Francisco, USENIX, pp. 133-146.

Skype Limited, 2006. *Guide for Network Administrators, Skype 3.0 Beta,* s.l.: Skype Limited.

Skype, 2012. *Forgotten your Password?.* [Online] Available at: https://login.skype.com/account/password-reset-request [Accessed 10 07 2012].

Skype, 2012. *Free Skype internet calls and cheap calls to phones online - Skype.* [Online] Available at: http:///www.skype.com [Accessed 18 07 2012].

Skype, 2012. *Help for Skype: Does Skype use Encryption?.* [Online] Available at: https://support.skype.com/en/faq/FA31/Does-Skype-use-encryption [Accessed 11 07 2012].

Skype, 2012. *How much bandwidth does Skype need?.* [Online] Available at: https://support.skype.com/en/faq/FA1417/How-much-bandwidth-does-Skype-need
[Accessed 10 07 2012].

Sripanidkulchai, K., 2001. *The popularity of Gnutella queries and its implications on scalability.* [Online] Available at: http://www.cs.cmu.edu/~kunwadee/research/p2p/gnutella.html [Accessed 18 07 2012].

Stoica, I. et al., 2001. *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications.* San Diego, ACM.

Stribling, J. et al., 2006. *OverCite: A Distributed, Cooperative CiteSeer.* San Jose, USENIX.

Stutzbach, D., Zhao, S. & Rejaie, R., 2007. Characterizing files in the modern Gnutella network. *Multimedia Systems,* 13(1), pp. 35-50.

U.S. Department of Commerce, N. I. o. S. a. T., 1995. Secure Hash Standard. *FIPS PUB 180-1*.

Wang, P., Sparks, S. & Zou, C. C., 2010. An Advanced Hybrid Peer-to-Peer Botnet. *IEEE Transactions on Dependable and Secure Computing,* 7(2), pp. 113-127.

Vaudreuil, G., 1996. *RFC 1893, Enhanced Mail System Status Codes,* s.l.: IETF.

Vuze, Inc., 2012. *Azureus, now called Vuze : BitTorrent Client.* [Online] Available at: http://azureus.sourceforge.net/ [Accessed 14 07 2012].

Zhao, B. Y., Kubiatowicz, J. & Joseph, A. D., 2001. *Tapestry: An Infrasctructure for Fault-tolerant Wide-area Location and Routing,* Berkeley: Computer Science Division (EECS), University of California.

Zhuang, S. Q. et al., 2001. *Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination.* Port Jefferson, ACM, pp. 11-20.

机电工程师, 2012. *Beacon Cache II 0.8.0.1*. [Online] Available at: http://www.jdgcs.org/Beacon2/gwc.php [Accessed 15 07 2012].