Ulysses - A new approach to maintaining connectivity

Manav Ratan Mital, K Vikram, Abhinav Gupta, Dheeraj Sanghi
Department of Computer Science and Engineering
Indian Institute of Technology Kanpur
Kanpur 208016, India
{manavrm,vikram,abhigupt,dheeraj}@cse.iitk.ac.in

Abstract

This paper discusses the design and implementation of Ulysses[1], an unstructured, decentralized peer to peer file sharing client. Contemporary peer to peer file sharing applications which have been deployed on the Internet have often been plagued by the problem of fragmentation of the network. Some of them have used the idea of installing special nodes on the network to facilitate the adaptation process triggered by unexpected events. Such monitor-like nodes often incur large maintenance costs and defeat the purpose of decentralization. We suggest a design that is more decentralized and that has been made with a focus on load balancing amongst the several connected nodes and maintain the network graph connectivity. The idea is to periodically exchange certain information between nodes that is used to maintain local connectivity in the network when some links fail. The protocol developed for the overlay network promises to be flexible and scalable enough to be deployed for large-scale use.

1 Introduction

Today, there is a growing interest in decentralized networks and distributed computing. A desirable network is one that avoids any single point of failure and in which nodes can organize themselves into an ad-hoc network that can be efficiently utilized for information interchange.

There are many peer to peer systems that are in use today. In particular the file sharing applications are quite popular. The early file sharing applications which were centralized included Napster (Fanning, 2000) and E-donkey (MetaMachine, 2000). One of the technical problems with their design was that their operation depended on a central server or a few central servers. These servers maintained the precise location of all the resources in the system. To locate a particular resource (a document, a media file or a software tarball, etc.) each node had to search for its location on one of these servers. Thus, in the case of failure of one or more of these machines, the network could instantly go down.

A complete decentralization, avoiding the burden on a few central servers seemed to be a solution to some of the problems. This led to the development of a decentralized peer to peer model as the one in Gnutella (Ripeanu et al., 2002), Freenet (Clarke et al., 2001), Morpheus (Streamcast, 2003), Kazaa (Sharman, 2003) and Groove (Groove, 2000), etc. This design eliminated the need for a central server, with each node running both a client and a server on the same machine. This architecture is similar in design to the DNS (Mockapetris and Dunlap, 1988) query mechanism, though there is no hierarchy typical of the DNS. The client on a machine is responsible for searching resources over the network and the server does the work of receiving requests for resources and forwarding the requests appropriately

[1] Named after the mythical Greek Hero, Ulysses, known for his explorations and adventurous voyages. In the mythological world he represented the spirit of freedom.

if it does not have them. Freenet is a peer to peer storage system which is symmetric and adapts to joining and leaving of hosts. As in other aforementioned systems, the search in this case is for cached copies and data may be redundantly stored over the network.

The problems, however, with the above applications are that they may lead to network fragmentation, or else incur a huge overhead of messages for connection maintenance. Any peer to peer system is a continuously evolving system, and needs to ensure global connectivity of its nodes. To this end, it should implement some kind of a *maintenance protocol* which takes care of the topology of the overlay network (Balakrishnan et al., 2002). In our discussion, we shall be talking about peer to peer systems in the context of file sharing.

An interesting instance of the aforementioned problem associated with peer to peer systems happened with the Gnutella network in August 2000. It was observed that the Gnutella network, once a totally connected topology, was fragmented into smaller networks which did not communicate with each other. This problem of fragmentation of the network, that we address here, still troubles contemporary applications (Anderson, 2001).

Another drawback, in particular with Gnutella, was that the query responses were returned along the same path as the one on which the query propagated leading to more traffic than actually necessary.

More recently, peer to peer systems have witnessed protocols like the distributed hash lookup protocols of Chord (Stoica et al., 2001) and CAN (Ratnasamy et al., 2001) which focus on the efficient location of data items and also promise to be fault tolerant to some extent. These distributed hashing techniques are theoretically sound, nonetheless they suffer from a significant limitation of all hashing techniques. They provide you with a particular file, only if you know the exact ID of the file. Searching for all documents of a particular type, which are common tasks in other file sharing systems, is difficult to implement on a distributed hash-table.

In this context, one would like to develop applications that are able to form a stable overlay network, that makes an efficient use of the available bandwidth and have fault-tolerance capabilities. The absence of a central node does away with quite a few problems but places certain additional responsibilities on all the participating nodes, as now the onus of maintenance of the network topology and services is on each of them. Also, the network must not go down because of the under-performance of a few errant nodes.

In this paper we describe Ulysses, a decentralized peer to peer servent (server+client) capable of sharing files across the network. The main issues that we have addressed in our protocol design, have to deal with ensuring the connectivity of the entire network at all times. This essentially provides for a kind of maintenance protocol, the need for which was discussed earlier. We use a minimum spanning tree algorithm (Aho et al., 1974), which runs periodically on each node and the result of the execution of this algorithm is the new set of connections that must be formed amongst its neighbours in the event of its *death*. Such information is periodically exchanged between adjacent nodes on the overlay network. Conceptually, this information is what a node would make use of in case of an abnormal change of the state of the system, to attempt to bring back the system to an ideal state.

2 Ulysses

Ulysses is the name of the P2P *servent* (server+client) that is described in this paper. It has been developed along the lines of Gnutella with some design enhancements in the protocol. The central issue that this application focuses on is the problem of loss of connectivity in the network topology, which may be introduced due to random node failures. It supports a maintenance protocol that ensures that a node remains connected to the entire graph at all times.

The client is a Linux based application, compiled using gcc version 2.96. It allows the capabilities of file-sharing, download and instant-messaging. It has an attractive gtk-based GUI for smooth operation. It is easily configurable and can be customized, though it can be run instantly after compilation, using the default configuration.

2.1 Overview of the protocol

The protocol is designed to perform on an overlay network of Ulysses nodes connected to each other over an IP network and uses both TCP and UDP protocols. The protocol was essentially designed to support an over-all **stable graph connectivity** and **load balancing** amongst the peer nodes in the network. There are ten types of packets that are used to communicate:

- *PING* - used to actively discover other servents on the network.
- *PONG* - sent in reply to the PING message by other servents when this node itself is ready to connect to them
- *JOIN* - sent to the nodes that this node wants to connect to.
- *JOINOK* - sent to the sender of the JOIN message if this node is ready to form a connection.
- *BUSY* - sent to the sender of the JOIN message if this node is not ready to form a connection.
- *ISALIVE* - sent periodically to check if its neighbour has not gone down.
- *ALIVE* - sent in reply to ALIVE messages to indicate that the node is still reachable.
- *SEARCH_REQ* - sent whenever the node wants to search for a particular resource on the network.
- *SEARCH_RESULTS* - sent by the node in reply to the SEARCH_REQ message if it possess the requested resource.
- *BYE* - sent to the neighbours when it leaves the network.

Each servent maintains the following configurable parameters:

- *LOAD_MAX* - is the maximum number of connections that a node can maintain.
- *LOAD_MIN* - is the minimum number of connections that a node tries to maintain.
- *LOAD_DES* - is the maximum number of connections that a node would actively want to establish when it joins the network.
- *MAX_TRY* - is the maximum number of attempts to connect to other nodes on the same subnet. After this it tries to connect to a peer broker.
- *KEEPALIVETIMER* - is the interval between two successive ISALIVE packets sent by a node. This is also the interval after which the minimum spanning tree operation (described in Section 2.4) is periodically performed.
- *PINGTIMER* - is the time for which a node waits for a PONG reply.
- *SEARCHTIMER* - is the time for which a node waits for a SEARCH_RESULT to arrive in reply to a SEARCH_REQ.

The protocol is described in detail on the next page:

- Node A, when it starts, tries to connect to other Ulysses nodes by broadcasting the *PING* messages on the network. The broadcast mechanism is essentially the same as in the Gnutella protocol. If there are no other Ulysses nodes in the same subnet, then a peer broker would be required to provide the first connection to this servent.
- Node B, that is ready to connect, replies by sending the connecting node PONG messages. The *PONG* message contains the degree for that node in the overlay network graph (which essentially means the number of nodes it is connected to).
- A then collects all the *PONG* messages and sends *JOIN* messages to LOAD_DES number of lowest degree nodes.
- B sends *JOINOK* message to A if, after joining, its degree does not exceed the value LOAD_MAX; else it sends A the message *BUSY*
- A ends up forming connections with all those nodes that reply to it sending a *JOINOK* message. This is a virtual connection maintained by making an entry in a table at A. If A is unable to connect to LOAD_MIN number of nodes, it

periodically repeats the *PING*ing process as described earlier. Here, we assume that A gets the requisite number of LOAD_DES connections.

- Whenever A wants to search for a file that is not present with it, it sends a *SEARCH_REQ* message to its neighbours which may then propagate the messages further. If any node has the required file with it, it establishes a TCP connection with A and sends the results encapsulated in a *SEARCH_RESULTS* packet. Then A may download the file using an FTP-like protocol.

- To prevent the *SEARCH_REQ* messages from going in loops in the (overlay) network every node remembers search requests that have gone through it. This cache is flushed periodically.

- Each node periodically tests to find out if its neighbours are alive by sending them *IS_ALIVE* messages. If the neighbour node is alive, it responds by sending an *ALIVE* message in return.

- If the node tested as above does not reply to the message, A removes it from its table of current connections and instead connects to the successor of the node that died (successor is determined by a procedure discussed below) using the *JOIN* message.

- Every t seconds each node performs a Minimum Spanning Tree (MST) computation, the details of which are discussed in Section 2.4. This computation enables it to decide a renewed graph topology formation that should take place in the event of its own "death". In effect, it decides what node each of its neighbours should connect to, to maintain the graph connectivity.

- After the computation is done, a neighbouring node is informed of its corresponding successor(s) in the *ALIVE* packet that is sent to it when demanded.

- When node A is about to leave the network, it sends a *BYE* packet to its neighbouring nodes. This packet contains which other node(s) the recipient node should connect to after A goes down. When a node goes down unexpectedly, the *ALIVE* message plays the resilient role.

2.2 Design of the application

The design of the application, as shown in Figure 1, is fairly modular. There are four components in a single servent that interact with each other and also with those on other nodes, exchanging information as and when required. These components are

- *Ulysses Client* knows how to communicate over the Ulysses protocol with other nodes. This is the component that is activated first when the application starts and sends PING messages, using UDP level broadcast. Once the node has made the desired number of connections, it then listens for user events from a graphical user interface. The possible user events are start searching, stop searching, send messages, refresh user lists, etc. The client generates appropriate Ulysses protocol messages over the network for each of these user events. For instance, when the user enters a particular search, the client builds a search packet and sends it to each of its neighbours and starts a timer waiting for search results from remote nodes. The stop search event would stop this timer. When the user sends a message, the client contacts the remote node and sends it the message according to the protocol. Additionally this component is responsible for sending IS_ALIVE messages periodically to check the availability of the neighbour nodes.

- *Ulysses Server* listens for requests from clients on other nodes. Search requests are first matched with resources on this node and if some resources match, their locations are sent directly to the node on which the request originated. If there is no match at all, this node can forward the request to its other neighbours. The server maintains a cache of recent search requests to prevent search requests going around in loops. The timeout value for this cache is configurable. The server also responds to PING and IS_ALIVE messages with PONG and ALIVE messages respectively.

- *File transfer server* listens for requests for files from remote nodes. It serves these requests by

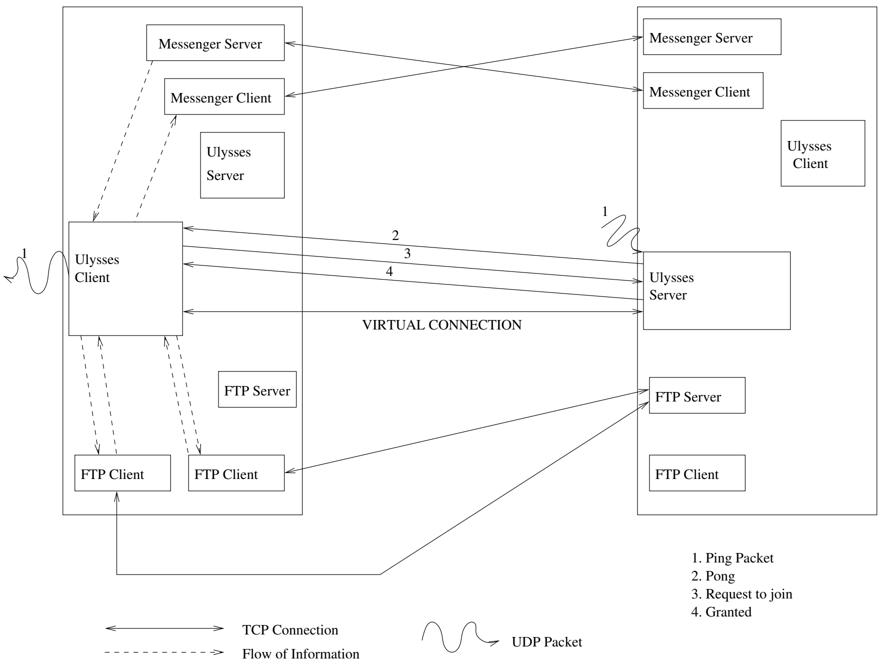

Figure 1: High Level Design of the Application

sending the file over an FTP like protocol. The messenger daemon listens for messages from other nodes. The messaging service present in the application makes the process of resource sharing much more convenient.

- *Messenger server* runs on a servent listening for messages from other servents. These are basically user messages that may be sent to the user operating the servent from the users of other servents. It provides instant messaging facility between the various nodes.

2.3 Uniform distribution of connections over the network

The protocol ensures that whenever a node connects to the network, it connects to the hosts with the least number of connections. This leads to a uniformly connected graph with a better balancing of load on the various participating hosts. This results in better resource utilization and bandwidth allocation. We can show, inductively, that the network will always remain globally connected with the arrival and departure of nodes from the network. This global connectivity will be maintained as long as it is possible to do so. As the base case, a single node is a single *connected component*. For the induction step, at a particular instant, let us assume that the network is connected. When a node joins this overlay network, it remains connected. When a particular node, say N, is about to leave the network, it executes the minimum spanning tree algorithm as explained in Section 2.4. This results in each neighbour of N getting to know which node it should connect to after N goes down. The algorithm is such that, if these connections are established, the network still remains connected locally. Thus inductively, the network remains as a single connected component when nodes go down and come up.

In practice, a better distributed graph also leads to improved search query handling, for if it were not the case, then a majority of the search queries would have to be routed through a single node, leading to greater overhead on that node.

2.4 Stable Graph Connectivity

Every t seconds, each node performs a minimum spanning tree computation for its immediate neighbours. Note here that this t is equal to the parameter KEEPALIVETIMER mentioned in Section 2.1 above. The load metric for the neighbours is defined in terms of their degree in the overlay network. The fewer connections a node has, the lesser the load on it. Thus, the spanning tree computation does the following

- Removes the node (at which the computation is being done) from the graph.
- Builds a least-cost tree amongst all the neighbouring nodes. This results in each node being connected to every other node by one or more paths. This tree computation is done by iteratively introducing an edge to connect an unconnected neighbouring node to the tree.
- The edges of the tree are so introduced that the node with minimum degree is preferentially raised to a higher degree. This is done by using a greedy algorithm.

The pseudo-code in Figure 2 explains the above algorithm:

After the above computation, the node informs each of its neighbours the IP address of the node that it should connect to in case the node itself fails. Periodically, it also keeps pinging its neighbours to ascertain that they are alive or not and updates their status accordingly. Each host pings each of its neighbour every t seconds. Now, there can be two cases:

1. A node leaves the network normally. Before doing so, it will inform its neighbours and then exit. This allows the neighbours to break the existing connection and instead, form a new connection with the respective successor advertised to them by the node.
2. The node simply dies without any notice. The neighbours will not notice this until they either ping that node (by sending an *ISALIVE* packet or when they need to send/propagate a query to it. Each neighbour will then form a new connection with the pre-advertised successor as in the above case.

Also, if a node has, at any point in time, less than *LOAD_DES* number of connections, then it periodically advertises invitations for more connections. The above steps ensure a well connected and balanced graph for a Ulysses network.

```
mst()
{
 //get the list of neighbours for the node
 ip[] = get_connection_list();
 //get total no of connections to this node
 total = get_no_of_connections();
 //get the list of loads  for the neighbours
 load[] = get_load_list();
 //sort the nodes in increasing order of their
 //degree
 sort(load, ip, total);
 //method to simply create a set from scratch
 make_set(SET);
 k=0;

 //the following loop initializes the set with IP
 //of the neighbouring node that has the lowest
 //load
 while(k<total)
 {
  //include node only if the load is non-zero
  if(load[k]!=0)
  {
   //include the first IP in the set
   include_set(ip[k], SET, load[k]);
   k++;
   break;
  }
  k++;
 } //the set SET has been initialized
//with the node with the lowest load

 // visit the list which is in sorted order
 for(i=k; i<total; i++)
 {
  //ignore nodes that have a zero load value
  if(load[i]==0)
   continue;

  //retrieve the minimum element in the set
  //this is the node with the least degree
  lowest = get_min(SET);

  //include the current node in the set
  include_set(ip[i], SET, load[i]);

  //connect the node to the least connected node
  connect_set(lowest->ip, ip[i], SET);

  //the method connect_set(n1, n2, set) forms a
  //pairing between the nodes n1 and n2 contained
  //in the set and increments their load value

  load[i]++; lowest->load++;
 }
} //mst computation over
```

Figure 2: Algorithm for MST computation

3 Experimental Results

Our system was run on a subnet on our local area network. Initially two nodes 1 and 2 form a virtual connection between themselves (see Figure 3). Node 3 then sends a PING message on the network and connects to node 1 and node 2. Node 4 after sending the PING message receives PONG replies from all the three nodes but connects to only node 2 and node 3. Node 5 also receives PONG from all four nodes. However it ignores the PONG replies of Node 2 and Node 3 because they already have three connections. Similarly Node 6 connects to nodes 1 and 5 and Node 7 connects to nodes 4 and 6.

Subsequently, to check fault tolerance of the system, node 2 was made to go down (see Figure 4). This caused the formation of a virtual connection between node 1 and node 4 as per the minimum spanning tree computation results last sent to them by node 2. When node 5 goes down, node 6 forms a virtual connection to node 4 and node 7 forms a connection with node 1. Then node 1 and node 7 go down leaving connections between nodes 6, 3 and 4 in form of a triangle.

To check the fault tolerance of the system in critical cases a hypothetical network of virtual connections was formed (see Figure 5). The unexpected shutdown of the node encircled in dotted line will be a critical case since this causes two disjoint meshes to be formed. However the network reconfigures as shown in the second part of the figure. Now the shutdown of the encircled node causes another critical case but the network again reconfigures to a stable state to prevent fragmentation into disjoint meshes.

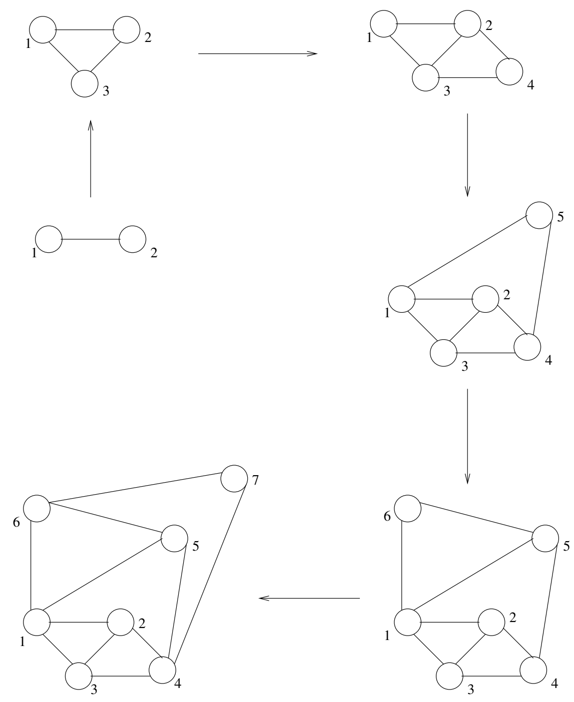

Figure 3: Formation of a hypothetical network

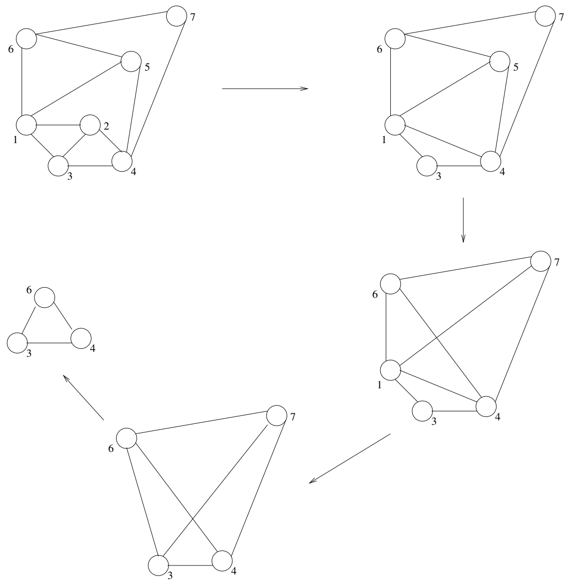

Figure 4: When nodes start disconnecting

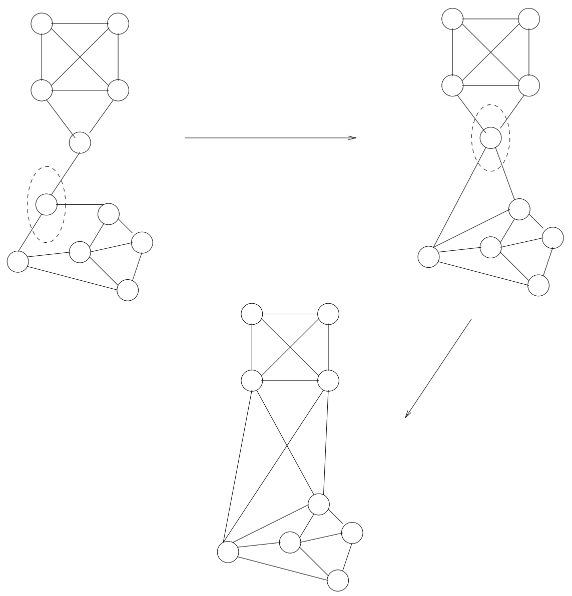

Figure 5: Demonstration of the fault tolerance of MST when a critical node goes down

4 Conclusions

In this paper, we have described a scalable, decentralized, fault-tolerant and self-stabilizing architecture that can be used by large scale data sharing applications. The broad vision of peer to peer systems is more than just sharing music files. They can be used to query information scattered over sensor networks. Such approaches can also be used to enable distributed data sharing over large computational grid infrastructures. There are a lot of algorithmic and architectural research issues involved in building these systems and our work attempts to move a step towards more elegant solutions to distributed system designs problems.

4.1 Future Work

There still remain a few design considerations that need to be worked out. There are some problems with our model which can be done away with by incorporating suitable features of other systems. For example, the broadcast mechanism can be made more efficient.

The protocol has been designed to take care of a large number of cases that may crop up when a node comes up or goes down in the network. But there are a few pathological cases where this protocol may fail to perform. These include link failures in which case, for instance, a node which may still be alive, seems to have gone down and another one is still assumed to be alive. In Figure 6 after the link shown fails, node B thinks that node D has gone down and tries to connect to node A as well as C, its successor. Now since, the connection between B and C must be formed on the same link as that between B and D, and that link has gone down, this clearly can't be done. These situations, the handling of which requires the knowledge of the topology of the underlying physical network, cannot be handled in our protocol.

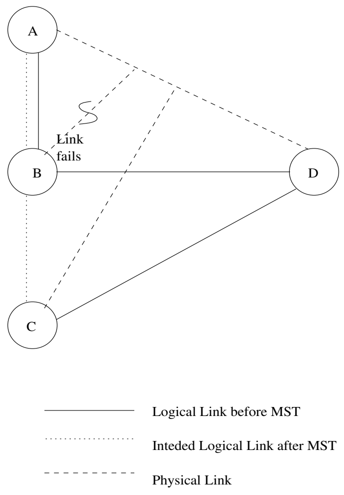

Figure 6: Link failure causes confusion to the node B.

References

A. V. Aho, J. E. Hopcroft, and J. D. Ullman. 1974. *Design and analysis of Computer Algorithms*. Addison Wesley.

Kelsey Anderson. 2001. Analysis of the Traffic on the Gnutella Network. CSE222, Term Paper, University of California, San Diego, March.

Hari Balakrishnan, David Liben-Nowell, and David Karger. 2002. Analysis of the evolution of peer to peer systems. In *ACM Conference on Principles of Distributed Computing (PODC)*, July.

Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. 2001. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009:46+.

Shawn Fanning. 2000. Napster. http://www.napster.com.

Networks Groove. 2000. Introducing groove. http://www.groovenetworks.com.

MetaMachine. 2000. E-donkey. http://www.edonkey2000.com.

Paul V. Mockapetris and Kevin J. Dunlap. 1988. Development of the domain name system. In *SIGCOMM*, pages 123–133.

Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. 2001. A scalable content addressable network. In *Proceedings of ACM SIGCOMM 2001*.

M. Ripeanu, I. Foster, and A. Iamnitchi. 2002. Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design.

Networks Sharman. 2003. Kazaa. http://www.kazaa.com.

Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM Press.

Networks Streamcast. 2003. Morpheus. http://www.musiccity.com.