Aljeda:
A self organizing, hierarchical, polymorphic, distributed peer to peer file sharing system.

Jesse Rabek, Damian Plummer, Aleksandr Berkovich

Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology

Abstract

This paper describes Aljeda, a second-generation, decentralized, peer-to-peer, distributed file sharing system. Unlike previous file sharing systems which make assumptions about client homogeneity, Aljeda uses a structured client/server system in which connected machines are delegated responsibility based on uptime and bandwidth. By utilizing a dynamic hierarchical tree structure, Aljeda provides load balancing, low latency, fault tolerance, and limited individual machine load.

1 Introduction

Current distributed file sharing systems suffer from a variety of problems resulting from incorrect assumptions about the properties of the clients. Namely, the first generation file sharing systems assume a certain level of homogeneity in regards to client bandwidth and participation in file sharing [1]. They also use poor network infrastructure which resulted in poor search results, excessive bandwidth usage, and segmentation within the network.

This paper describes Aljeda, a second-generation file sharing system that was designed based on experimental evidence of the shortcomings of previous systems [1]. Unlike previous systems, Aljeda takes advantage of the inherent heterogeneity of certain aspects of clients, namely bandwidth and uptime to share files more efficiently and effectively. By using a tree based structure and assigning different roles to clients based on their properties, Aljeda is able to reduce the amount of control traffic as well as limiting the diameter of the network.

There are four major design goals of Aljeda:

1. Minimization of traffic
2. Guarantee of high availability in the event of node failures
3. Minimization of network latency
4. Creating similar load responsibility between clients and clients acting as servers

Our current system does not ensure anonymity or security, although it is possible for security to be added later. Aljeda is easily extensible, and enhancements are mentioned in Section 7. As a beta system, all enhancements have been left out until the approach is proven.

Throughout the paper, node will be used to refer to any machine attached to the network. Also to avoid confusion later, clients are the only computers that actually serve files. In Aljeda servers are burdened with the responsibility of routing messages and maintaining network integrity.

Aljeda is currently being developed as a class project at MIT for the class entitled “Distributed Systems Engineering” under the direction of Professor Robert Morris and Frank Dabek.

2 Overview

Section 03 gives an overview of current systems and research in peer-to-peer file sharing systems and presents some of the drawbacks the current systems have that Aljeda hopes to overcome. Section 4 gives a complete description of the design of the network and describes its operation. Section 5 describes the implementation details of system. Section 6 evaluates the systems performance. Sections 7 and 8 provide the conclusions and future directions of the project.

3 Related Work

3.1 Napster/OpenNap

Napster, a file sharing program, is a service that uses centralized servers to match users to other users that have files, which match their queries. Unfortunately, having a centralized component in this manner made Napster vulnerable with a centralized point of attack, and legally responsible for copyrighted content shared by its users.

3.2 Gnutella/BearShare

Gnutella[3], engineered to be incredibly hard to block, is a response to efforts of institutions, such as colleges that had blocked Napster [2]. Gnutella essentially lets users search each other's hard drive. In theory, since each computer connects directly to every other, there is no involvement of central servers, no focal point, and hence no single point of failure. Unfortunately, any implementation of Gnutella can not really connect all users directly to one another, but instead makes use of overlapping clusters of machines. In this configuration, search queries are repeated on thousands of computers, which leads to an amount of traffic that can't be handled by the connection bandwidth of most users.

BearShare[9] is essentially a fix to the problems that plague the original Gnutella in terms of traffic and scalability. It routes most of the traffic through the most capable machines. These machines constitute backbones, which greatly speed Gnutella requests and allow less network congestion [2]. However, in reducing the congestion, BearShare has re-introduced centrality and reduced anonymity, making it and its users, vulnerable to actions similar to those used against Napster and its users.

3.3 FreeNet

FreeNet[8], Gnutella-like in its ideals, is more of a storage system than a file-sharing system. It is less susceptible to being shutdown because it encrypts all files and stripes them in chunks that constantly shift location [2]. Unfortunately, its inherent structure does not lend itself to be readily searchable, which causes it to be somewhat unpopular [2]. Also, since the system was originally designed with small documents in mind, constantly chunking large media files, we feel, is not practical on given a typical user’s bandwidth.

3.4 KaZaA/Morpheus

KaZaA[5], as well as Morpheus, a slightly modified client, is much like Gnutella in that it does not depend upon centralized servers, but unlike Gnutella, it uses a tiered system and is less bandwidth intensive. It is somewhat of a distributed self-organizing network, similar to BearShare where the backbone machines are called Supernodes. Each Supernode has the added responsibility of keeping indices of other machines in its neighborhood.

The backbone, however, is not designed to be dynamic and the same machines tend to be used repeatedly.

3.5 A Measurement Study of Peer-to-Peer File Sharing Systems

This is a study by Stefan Saroiu, P. Krishna Gummadi, Steven D. Gribble which monitored use of Napster and Gnutella, two systems which are at opposite extremes of the peer-to-peer model. It concluded that peer-to-peer systems must be very deliberate about delegating responsibilities since the peers in the system are really heterogeneous in bandwidth, latency, availability, and degree of sharing [1]. Some of the peers demonstrate clear client-like behavior while others demonstrate clear server-like behavior, so any new designs of file sharing systems should take it into account. Finally, peers tend to misreport information about their bandwidth, so to have effective delegation of responsibilities, either there should be built-in incentives for peers to be truthful or a peer-to-peer system should be able to directly measure the information.

4 Design

4.1 Tree Structure

The structure of the system determines the overall speed, mobility, and network congestion of the system. After evaluating other available systems on the market, we have decided that Aljeda should be optimized for the following criteria:

- Number of hops necessary for a client query request
- Polymorphism of the system (no permanent servers)
- Server migration time.
- Tolerance of the system to crashes of any of its servers.

Although the number of hops of a query request in Napster is constant, its network is static, and is therefore unacceptable. In Gnutella, the number of hops for each query request is on the order of n^2, so the system is not very scalable. BearShare and KaZaA\Morpheus are also much like Gnutella, though they have backbones, which speed up searches.

The main problem with all the above mentioned systems is that the servers try to store all the indexes of the machines to which they are connected. This approach is neither dynamic nor scalable. To illustrate, one machine sharing 10000 files would produce a 1 MByte index, assuming each file's information takes roughly 100 bytes. Thus, moving the indexes of just 30 machines takes three minutes using the full bandwidth of a DSL connection.

After carefully evaluating all previously available peer-to-peer (p2p) system configurations, we decided to use a hierarchical tree structure. The basic topology can be seen in Figure 1: Network Topology. System servers (i.e. machines that store and process index requests) act as nodes of the tree, while clients (machines that share and exchange files) are its leaves. This tree structure allows individual servers to hold only a fraction of all the client indexes. Client indexes are queried based on some criterion that spans the tree. In our implementation of Aljeda, we used simple alphabetical ranges to distribute file indexes. With this approach, the number of hops of a query request is on the order of $lg(n)$, where n is the number of machines in the Aljeda network. Although worse than Napster, this number of hops is significantly better than that of Gnutella and still requires no central server. In addition, the tree structure is dynamic. For example, if an individual server is overloaded with requests or holds too many indexes, it may respond by splitting into two servers in order to manage the load. In making this split, the server indirectly requests that one of the clients in the system becomes a server and shares the system load. As in other p2p systems, any user machine can either be a client, a server, or both.

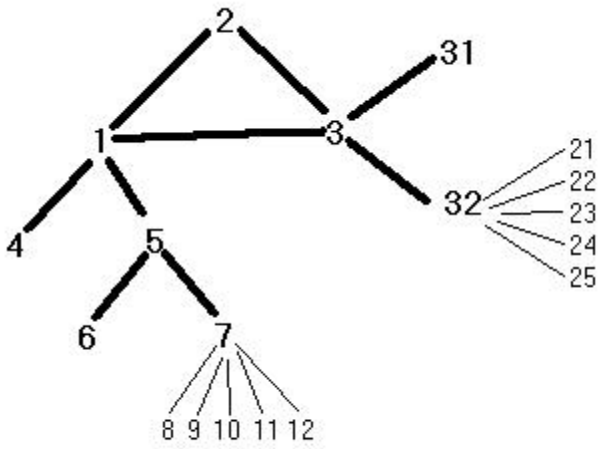

Figure 1: Network Topology

Referring to Figure 1, Aljeda's network structure consists of a top ring of servers (1,2,3). Each element of the ring spawns a tree (5, 32), whose leaves are clients of the system. All members of the ring are connected to each other and their respective children, while each of the other nodes in the system is only connected to its children and its parent. Each non-client node is responsible for a certain range of indexes and each of its children is responsible only for a unique subset of its parent's range. Thus, a parent may serve range X, and its n children will serve ranges X_1, ..., X_n, where $X_1 \cup ... \cup X_n = X$ and $X_1 \wedge ... \wedge X_n = 0$. A concrete example of how this works can be seen in Figure 3: Keyword Request.

4.2 Node Types and Roles

Figure 3: Keyword Request gives a detailed picture of the node types that appear in the network . The following sections describe the purpose of each of these nodes and how they interact with the other nodes.

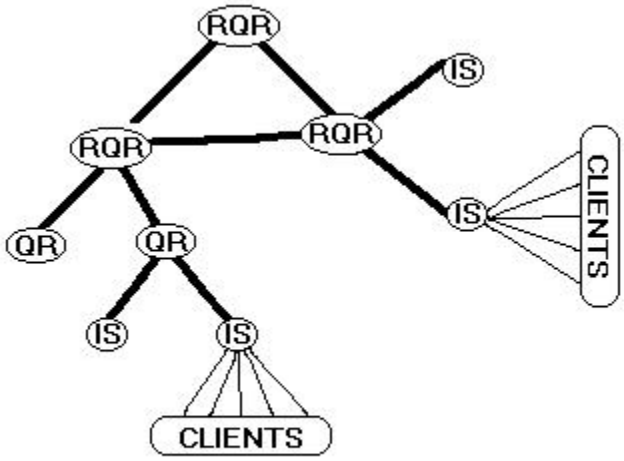

Figure 2: Node Types

4.2.1 Index Server

Nodes whose immediate children are leaves (i.e. clients) are called Index Servers (IS). They serve a dual purpose.

First, an IS serves and processes all the requests coming from its children, the client leaves of the tree. If a client wants to find out which machines contain a certain keyword, it would request this information from its parent, an IS. This IS, in turn, would pass the request up to its parent for further processing. When a client initially connects to an IS, the client uploads its own index, a list of keywords made from the names of its files. This information is also passed up the tree for further processing. The Index Server is the only server that a client interacts with in the system. The rest of the system is totally hidden from it.

Second, an IS stores all the <keyword, machineList> pairs, for the range it serves. An IS is also responsible for keeping the information up-to-date. It does so by using various update and expiration techniques. If an IS receives a keyword request, which falls within its range, the IS would immediately return a corresponding list of machines directly to the client that originated the request.

4.2.2 Query Router

Servers that are not immediate parents of clients are called Query Routers (QR). They differ from Index Servers in that they do not store any indexes themselves. QRs do not server any clients directly, and each QR's children and parent are always servers. Since QRs also have ranges, their primary function is to serve ranges, as well as to forward requests. Upon receiving a lookup keyword request, a QR merely redirects the request to an appropriate machine: to its parent, if the request came from one of its children and needs preprocessing, or to the appropriate child, if the request was received from a parent, and therefore falls within the QR's range.

4.2.3 Ring Query Router

The nodes that form the top ring that spans all trees are called Ring Query Routers (RQR). Their functionality is similar to QRs. Like QRs, RQRs do not server indexes. However, there are several important distinctions. Unlike the other servers in the system, RQRs by definition do not have a parent and are instead connected to all other members of the ring, their brothers as well as their children. RQRs handle the biggest load of the system, since their ranges are larger than any of their children. RQRs process all the client requests that they receive through tree forwarding and distribute them to appropriate brothers, according to their ranges.

4.2.4 Client

Clients are the users of the system. They do not serve other machines' indexes in any way. Clients transmit their index to the IS parent upon a start-up. After that they use the system for keyword requests and to contact other clients for file transfer requests. Clients download files from other Clients, but not from Servers. The main function of clients is sharing and exchanging files with other Clients of the system.

4.2.5 Duality

During the tree formation process, clients can become servers and then again clients at any time. That should not, however, hinder the regular use of the system (i.e. the ability to download files). We use the following approach to resolve this problem. When a Client becomes a server, it does not lose its downloading capabilities. It merely moves into a dual state: one process in the machine runs a Client, and the other one, a Server. In order to keep the bandwidth in control and encourage users not to log off, while being servers, the machine turns off any file uploading, while being in a dual state.

To keep the described tree structure intact, the duality gets implemented the following way: the Client and the Server processes become two separate virtual machines that reside independently in two different locations in the system structure. For all other machines these two processes are regarded as two different physical machines.

4.3 Network Actions

The following sections summarize the behavior of the network and its components.

4.3.1 Client Actions

4.3.1.1 Joining

When a new client wants to connect to the system, it first connects to any machine in the network. Its connection is forwarded up the tree until it reaches one of the Ring Query Routers. The RQR then sends the connection request down its tree by choosing a random child at each level to which to forward the request. Finally, the IS that receives the request picks the client and declares it as its new child.

Upon connection to an IS, the Client sends its index, a keyword list. The Index Server then passes it up the tree, until it reaches a RQR. There it gets separated and sent to other RQRs with the appropriate ranges. The RQRs, in turn, send the indexes down their respective trees, so that they can reach and get stored in the appropriate IS.

Note that the above only works if the client knows a machine that is already part of the network. In the case where the client is unaware of any machine already part of the network, a DNS server could be used that would resolve a static name, such as Aljeda.net, to one of the RQR and the joining process could continue as normal.

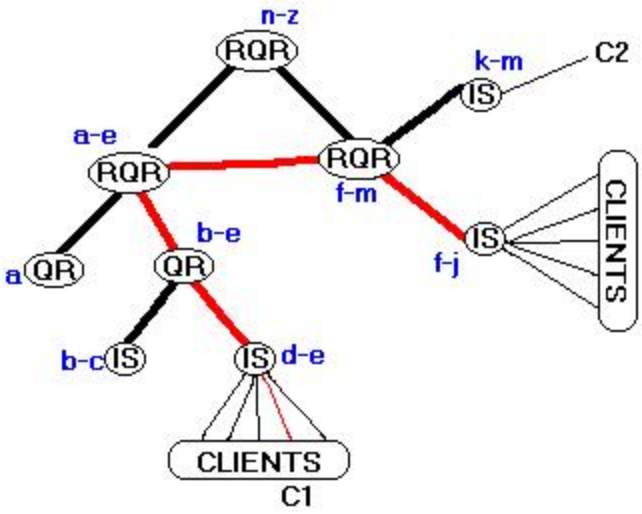

Figure 3: Keyword Request

4.3.1.2 Keyword Request

Suppose a client wants to find out which other clients have files associated with a particular keyword. It would send a Query Request. This request behaves very similarly to the startup index forwarding. The entire process is depicted in Figure 3: Keyword Request.

First, the request reaches the parent of the client, an IS. The IS passes the request up the tree to RQR. RQR examines the keyword and sends it to the RQR with the appropriate range. The latter forwards the request down the tree to the IS with the range that contains the keyword. Finally, this IS replies directly to the Client that originated the request, if the <keyword,machineList> pair was found in its index table.

4.3.1.3 File Transfer

When a Client has an address of the machine associated with a desired keyword, it talks to this machine directly. First, it sends a request for the names of all the files, associated with the keyword. Then it chooses a desired file from the returned list and requests a particular file. Finally, the remote machine sends the file requested.

4.3.2 Migration

The structure described in Section 4.1 is only a static snapshot of the system. In reality, no machine has a permanent assigned task. Our system is designed to be dynamic. The migration process plays a central role in maintaining this property.

The migration process is invoked in the following three cases:

1. After a certain predefined period of time, a server has finished its server duties and wants to retire (i.e. wants to become a conventional client). In that case, the machine transfer is complete internal state to a newly found replacement.
2. The server experiences too much network load. A situation that the servers close to the ring encounter frequently. It that case, the server invokes its load-balancing mechanism and transfers completely or partially its internal state to the newly found replacement. See Section 4.3.4 for more detail.
3. An IS has reached its predefined capacity for holding indexes. In that case, the server makes a brother and transfers him half of its range and corresponding indexes or spins off two new children, gives them half the range each, and becomes a QR itself.

When a machine decides to invoke a migration process, it first picks a target machine. The possible candidates can only be picked from the broadband clients, which are not in the dual state. The way our system handles this task is the most intuitive one: the server picks a random eligible client-leaf by passing a *FindNewServer* request down its tree. The client that received such a request enters dual mode and reports back to the origin of the request with *FindNewServerFound* message. The origin-server then passes all the appropriate information to the destination-machine: the family (children, brothers, parent, and uncles), the serving range, and, for IS, the index file.

4.3.3 Tree initiation and growth

The minimum requirement of the system is having two broadband computers and a permanent DNS server. One broadband computer becomes a RQR, and the other one connects to the ring as an IS. Both initially serve a complete alphabetical range. Then the system grows and lives according to the following scenario:

- The clients always connect to a randomly chosen IS. See Section 4.3.1.1 for more detail.
- Only broadband clients can become servers.
- When the server experiences too much network load, it creates a new brother (see migration section) and assigns him half of its serving range.
- When an IS reaches its predefined capacity for holding indexes and has less then five brothers, it creates a new brother and assigns him half of its serving range and indexes. Otherwise, it spins off two new children, gives them half the range each, and becomes a Query Router itself.
- When a server loses a connection to its parent or brother, in the case of Ring Query Router, it invokes the Fault Tolerance mechanism. See Section 4.3.5.
- When a client looses a connection to the system, it simply asks a DNS server to connect to the system again.

- After a certain predefined period of time, each server retires. In that case, the server invokes its Migration process.
- Each client that is not in a dual state can be called to become a server.

4.3.4 Load Balancing

We have made efforts to ensure that the load per individual node in the system is reasonable to tolerate. Our two most notable methods are: 1) noting the dual nature of each node and making sure that no node serves files, while it is serving ranges or indices; and 2) enforcing time based transitions or migrations, so that no node will constantly have the burden of being a server.

Whenever a node of Aljeda is operating as a server, serving ranges or indices, its client module will disable its ability to serve files to other clients. This reduces the potential of a double serving duty on any one node.

Besides allowing for a dynamic tree structure, migration allows a node to transfer its serving duties to another node after a specified time period, at which point the server sends a message via its children to find a client to serve as its replacement. The server picks the first eligible client that makes contact. The server then transfers its range if it is a Query Router or Ring Query Router and its index as well if it is an Index Server. After a successful transfer, the old server tells its children to reconnect to the new server, and the new server then informs its new family of its existence.

4.3.5 Fault Tolerance

Our system is able to survive various types of errors, in particular the case of client/server death and the case of lost connections.

No special handling is needed for the case of a client's death except that the first querying machine that realizes the client's demise will inform the referring Index Server of the client's death. This will prevent the Index Server from further indicating this client as a viable connection point to any other querying clients whose query matches a file in the now dead client.

Whenever a server dies, its death is realized upon a child's failed attempt to send that server a packet. In this case, there is immediate restructuring of that server node's tree. The restructuring is initiated by the dead server's children, who trickle a “server replacement request” message down their respective sub-trees by picking one of its *n* children at random. These children in turn pick only one of their children to which to propagate the message. At the lowermost level, *n* clients will have received the message. Each eligible client then contacts the children's grandparent in the case of a common Query Router death or a pre-selected uncle of the dead server in the case of a Ring Query Router death. The grandparent or uncle chooses the first of the eligible clients to connect to it and subsequently. The appropriate family members are informed of the replacement.

Aljeda guarantees transparent inter-node connectivity through its connection layer, which maintains and reinitiates connections even in the event of reset and lost connections.

5 Implementation

5.1 Module Organization

Figure 4: Inter-module Communication gives an overview of the modules in the system. The controller object is responsible for starting, stopping, and managing the client and server objects,

as well the lower level message handling object. Client and server objects communicate with the controller object regarding changes in state and with the message handler object for external communication.

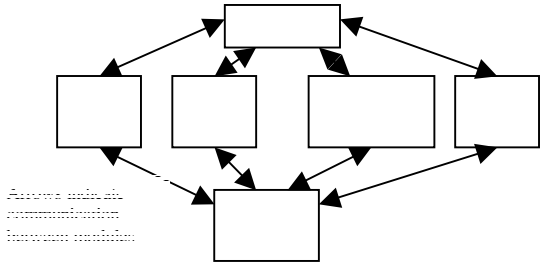

Figure 4: Inter-module Communication

5.2 Code Details

Aljeda is completely implemented in C++ and will currently compile on FreeBSD although with some minor rewriting, the code can be ported to any operating system capable of compiling C++. We used the asynchronous libraries of the Self Certifying File System [10] in order to avoid concurrency issues. Asynchronous programming requires the use of callbacks since all IO calls must be non-blocking. We extended the use of callbacks to encompass communication between the thereby reducing dependencies and increasing flexibility. Modules set callbacks in the message handler for each type of packet they are interested in receiving.

5.3 Protocols and Communication

The connection layer of Aljeda consists of message handler that maintains the active connections and is responsible for marshalling and de-marshalling Aljeda uses standard TCP/IP sockets to communicate between different machines. In order to communicate between machines, the client or server objects create a data packet consisting of structured data. In order to send the structured data over the network in a linear buffer, all of the objects used in Aljeda are capable of serializing and deserializing themselves. We explicitly avoided RPC and other mechanisms to avoid platform dependency.

6 Evaluation

In order to evaluate our system, we set up both a static network by explicitly creating the machines and connections, as well as a network that grew by having machines connect one at a time.

Once we ensured that the dynamically grown network self organized properly, the completely initialized network was used for most of the testing in an interest of the speed at which it could be created and destroyed when testing multiple facets of the system.

For each design goal below we present a brief summary of how our system achieved it.

6.1 Minimization of Traffic

In an Aljeda network of n nodes, the system was able to maintain a network diameter through its growth algorithm of around the predicted $\Theta(\lg n)$. This provides a nice limit on the distance a packet can travel within the network.

Search traffic was minimized since packets only travel up to the ring query router and back down to the appropriate index server and then all subsequent communication is peer to peer. The network does not need to rely on a flood based approach of any sort.

The need for keep-alive messages between nodes was eliminated by detecting node failures on packet send. Also, all network healing is all taken care of locally and requires a average of $c\lg(n)$ packets, where c is the number of child nodes the dead nodes has and $\lg(n)$ is the depth of the tree.

6.2 Guarantee of High availability in the event of node failures

Several different tests were performed to evaluate the robustness of the network. The tests involved the evaluation of the system's response to server node failures. Specifically we tested if the system recovered successfully and the specific effects of the death. In each set of tests, a different type of server node was killed at a random time. The results of killing each server node type appear below. Note that if a client dies, it simply needs to re-connect to a ring query router to be part of the network again and entries in index servers around the network are still valid since they reference an IP and not a location within the network. Therefore, tests involving its failure are not included below.

6.2.1 Query Router

As per the failure strategy mentioned in Section 4.3.5, the children successfully located a replacement for the server among the lower level clients and this replacement informed the dead query router's parent. However, when the replacement strategy detailed previously failed because the clients are not present or the connections were reset, the system does not attempt to retry and the user must manually reconnect their node to the network by re-contacting a ring query router.

6.2.2 Ring Query Router

The results of killing a ring query router had similar results as the query router. However, due to its location within the network, the death of a ring query router stalled communication of an entire sub-tree with the rest of the system until a replacement was found.

6.2.3 Index Server

The death of an index server appeared to have minimal impact on the system. Clients could not locate the files whose indices were stored on the dead index server even though they existed on the network. However, this problem could easily be solved through replication, overlapping indices, or other methods. Clients that were connected to the dead index server simply reconnected to the network as they detected the fact that their parent was dead.

6.3 Minimization of Network Latency

Network latency was minimized in general by the way in which the network grows and the number of hops necessary to reach anywhere on the network as mentioned above. This would not be to any advantage if the entire network had to be crawled to obtain search results or perform any sort of index distribution. However, Aljeda's highly ordered network proved to be able to perform searches and index distribution with $lg(n)$ network hops.

6.4 Creating Similar Load Responsibility for servers and clients

In a pursuit to equate bandwidth utilization by not having servers serve files, we discovered that in a large Aljeda network higher level servers received large amounts of traffic even when the ring was expanded to a fairly large diameter. The cumulative traffic of every machine in the

network routing packets though the top level servers proved to be much more than any particular client simply serving files.

7 Limitations and Future Work

Even though the current version of Aljeda implements only the basic functionality described in this paper, it can be easily extended to include other features. Below we included features that could be easily added to the system broken down by category.

7.1 Reliability

One of the main drawbacks of the system now is that there is no overlap in the query ranges of index servers. A major improvement would be to replicate indexes or at least overlap them to limit the impact of failure of individual server nodes. This would require only slight modification to the query router module, since the index servers with overlapping query ranges would not have to be aware of one another.

7.2 Security

Machine to Machine to security would vastly reduce the susceptibility of the system to attackers. Currently the system could be easily be brought down by lying to machines about family members and indexes among other things.

Encryption at the packet level could also be added to further protect users.

7.3 Latency and Throughput

To increase throughput in the system, both for queries, downloads, and control messages, compression could be utilized to reduce packet sizes. It is debatable whether or not this would increase the performance of the system since individual query and control packets are small already, and the media files being transferred most likely have fairly high entropy due to the compression algorithms in use already. Implementation is simple as only the message handling module would have to be modified to support compression.
Another client side addition that would increase both reliability and throughput is striping. This would enable a client to request different parts of a file from multiple machines. This would help throughput only if the network bottleneck is occurring at the client's end that is serving the file. It would always improve reliability in the case of multiple clients serving the file though since the failure of individual clients would not disrupt the downloading of a file.

8 Conclusions

Overall the system performed well. Server failure and migration did not compromise the network integrity. Aljeda provided fast, effective, exhaustive searches while minimizing strain on the network.

Some of the improvements in Section 7 should be implemented before the system is used. Namely, the system can easily be brought down by malicious users right now as there is no authentication between machines.

Also, there is some concern that as the network grows that the Ring Query Routers would become overwhelmed by the raw number of packets needing to be routing. However, the machines that comprise the backbone of the BearShare and KaZaA systems seem to be able to handle the traffic from their sub-trees, so it is still left to be seen whether Aljeda will have problems with scalability.

References

[1] S. Saroiu, P. Gummadi, S. Gribble, "A Measurement Study of Peer-to-Peer File Sharing Systems," [Online Document], Jul 2001, Available HTTP: http://www.cs.washington.edu/homes/tzoompy/publications/tr/uw/2001/uw-cse-01-06-02.html

[2] C. Mann, "Taming the Web," *Technology Review*, [Online Serial], Sep 2001, Available HTTP: http://www.technologyreview.com/magazine/sep01/mannall.asp

[3] Gnutella Website: HTTP: http://www.gnutella.com

[4] Napster Website: HTTP: http://www.napster.com

[5] KaZaA Website: HTTP: http://www.kazaa.com

[6] Morpheus Website: HTTP: http://www.musiccity.com

[7] OpenNap Website: HTTP: http://opennap.sourceforge.net

[8] FreeNet Website: HTTP: http://freenet.sourceforge.net

[9] BearShare Website HTTP: http://www.bearshare.com

[10] Self Certifying Files System Website HTTP: http://www.fs.net