medium.sized peer.to.peer database.system for secure.data.transfer

research&thesis.by
rob.haining

advising.by
dr.errin.fulp

2004

presented to the department of computer science
at wake forest university for consideration
of the distinction of honors in computer science

table.of.contents

section.0
abstract

Databases have historically resided on a lone, central server; however, this design is not a scalable, robust system. The single server database represents a single point of failure and vulnerability. While data-sharing systems based on the peer-to-peer (P2P) networking model have gained widespread popularity in recent years by providing a more scalable and robust solution, entries in the P2P system are clients and servers; therefore, data is not centrally located on one machine. Unfortunately, P2P systems lack the trustworthiness necessary for certain types of applications. Therefore, creating a database system that is secure, distributed with replication, and scalable remains a challenging problem.

The proposed system addresses these issues by combining aspects of traditional database systems, P2P networks, and public key security. A logical ring of computers allows database actions that result in direct information transfer in a P2P fashion. In contrast to many P2P systems, security is provided via a ring of trust where members of the "Ring" need only trust their immediate neighbors. As a result, the ring of trust allows users to transfer public keys securely (via their neighbors). The major advantages of the proposed system are its robust P2P networking, its increased scalability over traditional databases and, most importantly, the presence of security in critical parts of the system. While the proposed solution is not fully scalable, the system's layer of security on top of the basic network adds a facet to distributed databases that is typically unavailable.

section.1
introduction

Today, the majority of websites rely on a simple server-client paradigm. To expound on this a bit, there exists one server for the website, and, via a web browser, the end-user (client) can retrieve pages, images, and other various and sundry data from the server. It exists to serve you, and it does it well enough for most use. An easily accessible example of this model is the Wake Forest University (WFU) website. There is one server that exists on the campus of WFU, and all the pages and other data are located on that server. When you visit http://www.wfu.edu/, you retrieve information only from that particular server.

Traditionally, databases follow this model as well. There is one server on which the database resides, and there are various clients that can connect to the server via various means and retrieve whatever data resides within it. A simple example of a database is the Wake Information Network (WIN) at WFU. WIN consists of a single server managing a database. Students, staff, and faculty are able to view, modify, or delete various items (based on their privileges) all via their personal computers. As you can tell, this follows the client-server approach, where all the data is located in one database on one machine. One cannot depend too much on this central server model. If that server fails, then all the data becomes inaccessible. In addition, that central server has to regulate every piece of data that comes in and goes out, which becomes a large burden on the machine. As a result, the performance decreases as the number of requests increases. Thus, the centralized approach is simply not scalable. One could purchase faster servers, but this still does not make the system scalable. Increased traffic, decreased reliability, and increased restriction on data are just a few of the problems associated with this model. Recently, a new model for content sharing has been introduced in the form of peer-to-

peer (P2P) networks. The P2P paradigm has started shifting the focus of research and application development in recent years. The basic idea behind this phenomenon is a distributed method for storing and accessing information. Instead of having a single server with several clients connecting to it, every computer becomes a server in itself. Hence, computers can connect to each other and through each other. The line between client and server has been blurred in this new model.

To follow the line of file-sharing, the first program to gain mainstream popularity was the now infamous MP3 file-sharing program, Napster.[1] While you still logged on to a central server in order to find other people from whom you wanted to copy songs, the actual data transfer was achieved via a peer-to-peer connection. The problem here was still the reliance on the central server, in addition to the lack of a scalable method for finding information. The next level up from this removed the central server entirely; some such programs were Gnutella, Morpheus, and Kazaa, which are "true" peer-to-peer applications.[2] In other words, you become a node in a large mesh of other users that are sharing files. There is no true central server that has any kind of control over the rest of the network. Each person is responsible for sharing only files of a legal nature and is thus the only entity able to be held responsible for illegal actions. There is a difficulty here in how to approach the search problem – if the network follows no particular pattern, how does one systematically search the network?

The idea behind this project lies in the coalescence of peer-to-peer networking, websites, and databases. A database has been created that is distributed amongst several servers (with replication), where each server is responsible for maintaining its own data. Traditional ACID properties will be maintained. The computers are then connected using the P2P networking

[1] Napster.com
[2] Gnutella.com, Morpheus.com, Kazaa.com

scheme with a logical ring as the graph on which the network is based. This essentially means that each server is only aware of two other machines (its "neighbors") at any given time. The entire database is accessible via web pages located at every server on this Ring. Administrative functionality is also available via web pages on each server.

If any computer fails for whatever reason, the Ring will automatically become whole again, through a self-healing process. Thus, the system will never be down, unless every computer on the Ring is down. Thus, one already has increased reliability. Each node is responsible for adding, deleting, and updating the data on its own computer, so the effects of censorship are lessened and a wider diversity of data ensues. Now, the traffic is dispersed amongst several computers (rather than just the one), so one machine is no longer being hit with so much traffic at once.

In order for this current system to function as expected, one must assume that all interactions with the Ring and any local database are through the web scripts provided. In order to update data on the Ring or flush it out, one must be the original author (identified by email address and password) and also be initiating the update from the server on which the data was created. Each piece of data is uniquely identified by the originating server, the author, the date and time of creation, and the date and time of the most recent modification. Thus, it is not necessary to implement measures to ensure clock synchronization among nodes on the Ring.

The goals of the proposed P2P-based database system are to make as secure yet optimal a system as possible, while fulfilling the standards of both databases and peer-to-peer design. The system will be shown to be distributed with replication, robust, secure, and fast. By doing so, it will be proven that the Ring is worthy of your time and attention.

###

section.2
general.assumptions

The following have been assumed.

I assume that no malicious users exist within the proposed system.

I assume that all interaction between a user and a database will be via the web scripts provided.

I assume that all interaction between any two nodes will be via the ring of trust.

I assume that there will never be less than three nodes in the Ring.

###

section.3
motivation

The motivation behind this project relates to college radio stations. Being the Station Manager of WAKE Radio (the student voice of Wake Forest University) and studying Computer Science at WFU, my interests naturally lie somewhere in between music, radio, and Computer Science. One idea led to another, and what came out of it was this project. The project originally stems from (a) a lack of communication between college radio stations and (b) the struggle of a local musician.

Almost every college radio station has a section of their album stacks devoted to local music. The problem here lies in the fact that the primary way that other stations can find out about this music is if that artist gets a break on a national level – currently this is limited to getting signed by a label with decent promotional capabilities or through publications such as the College Music Journal (CMJ). As far as I am aware, there is no system in which college radio stations can easily communicate with each other about the artists local to them that are worth listening to. This project hopes to remedy this situation.

What this system allows stations to do is to post reviews of local artists to their individual websites. The added bonus is that you can also search the entire database, which would consist of several other college radio stations. Thus, you would be able to read about whatever kind of music you like from bands you have never heard of that are known only in a small part of the country (or world!).

This problem can be viewed as a "medium" sized problem – while clearly this does not necessitate the enormous scalability of such P2P systems as Kazaa,[3] it does require the security presented in this solution. Additionally, the problem would not be solved well using a single

[3] Kazaa.com

server due to the immense distribution desired. This is one example of what the system developed can be used for. However, other implementations would also benefit from this type of system.

The rest of this paper will consist of a slight expounding on previous work in this field, where traditional websites and databases will be discussed in further detail, as well as peer-to-peer forays into these areas; an unabridged explanation of the system design, including why it is superior to all others; a discussion on the implementation; and finally, some concluding remarks, along with ideas for future research. I hope you enjoy the ride.

###

section.4
previous.work

As previously described, the major advantage of the traditional database lies in its singularity. An administrator only needs to worry about data on a single server and its connections to various clients. Below is a visual illustration of a simple client-server scenario that could be a traditional database (where every dot is a different client that is interacting with the server):

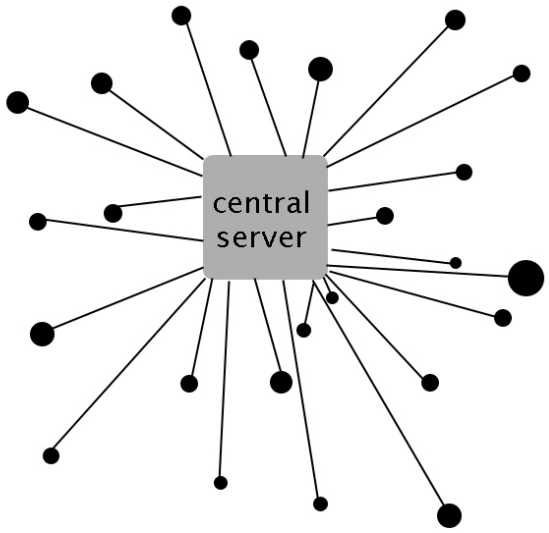

Figure A

If the controlling entity is interested in ensuring the quality of specific data and/or controlling where that data is served, this model is ideal. At any given time, one can easily know everything that is currently located on the database, who has retrieved data and what that data is, as well as who has changed data and what that data is. As a result, this model is fairly strong. One can perform queries, additions, updates, deletions, et cetera fairly rapidly.

Databases adhere to what is commonly referred to as the "ACID" test, which is an acronym for Atomicity, Consistency, Isolation, and Durability.[4] Atomicity refers to the ability to ensure that if one part of a transaction fails, then the entire transaction fails. This prevents partial transactions from taking place within a system. Consistency is the capacity to uniquely identify the various pieces of data that make up the database. Isolation signifies the protection of the shared resource known as the database: in other words, when a transaction occurs while another transaction is currently taking place, the newer transaction will either be locked out or will be given a view of the database from before the older transaction began. If the database crashes but there is a backup somewhere, then the database fulfills the Durability portion of the test. With these simple yet complete factors, one can guarantee the quality of a database design.

However, the traditional database does have its faults, which incidentally are also the product of its singularity. As said before, all the data is located on one server. What happens if the server fails? This could be a local problem with the machine or the database itself, or it could be a networking problem – perhaps clients are unable to make a connection to the server for some reason. In any case, when these problems happen, access to any data in the database is (temporarily) unavailable. Another problem with this model is the fact that every time a user requests data, adds data, or likewise, the single server has to handle 100% of the load.

Since the inception of the Internet, people have relied on the simple client-server approach to pass data between each other. However, P2P systems allow every personal computer to become both a client and a server simultaneously. Below is a visual illustration of a sample model (where every dot is a node in the P2P system, and the lines are connections between them):

[4] Parker, Jason

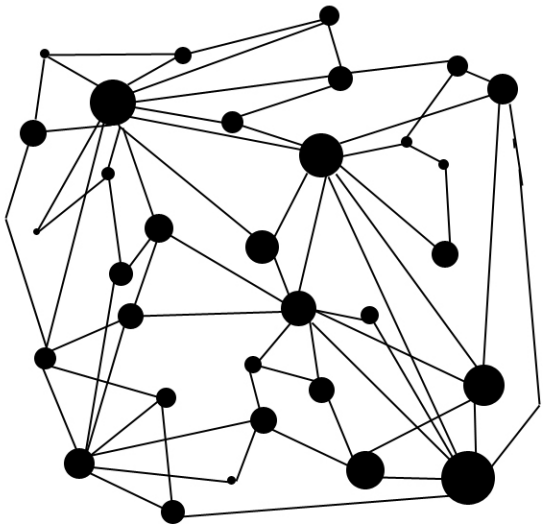

FIGURE B

One of the first popular applications that used the P2P model was the aforementioned Napster.[5] Ignoring the questionable legality of the software for a moment, Napster was a veritable breakthrough in standard networking models. Upon opening Napster, one would be connected to a server in the traditional sense, but also connected to that server would be millions of other users. While the server handled initial connections (a potential bottleneck), all file transfers were done directly between the users, hence the term peer-to-peer. Thus, the servers did not have to handle the immense load of either hosting such copious amounts of data, nor did the file transfers have to pass through them.

Several programs resembling Napster followed the model fairly closely, with a variety of subtle changes, but the next major development in the P2P world was not till the advent of the Gnutella[6] network, which completely eliminated any central server to handle the initial connections. With Gnutella, one's computer was not only the client and the server, but also

[5] Napster.com
[6] Sort of like Nutella®…think less chocolate and more music.

acted as a central server. When Gnutella was opened, the computer would be connected to another computer like one's own and would join the more "pure" P2P network of computers.

As it is oft to happen, it took a few years for Academia to catch up. By 2001, Dabek and then Stoica had developed the idea of the Chord system.[7] Basically, there are *n* users, uniquely identified by IP address and arranged in a logical ring. Whenever a user adds a new piece of data to the system, the index for the data (a short description and location) is stored at a random node. By using strategically set lookup tables, the Chord system minimizes lookup time. However, there are some disadvantages to this system. The Chord uses a high amount of overhead to store those same index tables to keep track of where other nodes are located. In addition to the requisite space on each machine, updating these tables adds overhead in terms of flooding the system with messages of people coming and going.

One major secure P2P system has been developed by the same folks that brought you Kazaa. This new system is called PeerEnabler™ and is developed by Joltid.[8] Basically, the system starts with licensed folks publishing data along with a signature file that can authenticate the data as published by that particular author. The data is originally on a webserver, but when end-users start downloading it, they automatically share it on their own computers, thus creating a P2P system. There are three layers in PeerEnabler: the client, the super nodes, and the "Network Management Server (NMS)." Security is handled using public key encryption methods to sign and authenticate data. In order to ensure that folks are who they say they are, keys are only distributed by a central authority, operated by Joltid. In doing so, they take away possible autonomy, as well as the beauty and elegance of a P2P system without any sort of centrality to it. Additionally, the NMS manages the super nodes. One cannot leave and enter

[7] Tanenbaum, Andrew S.
[8] Joltid.com

without knowing and acknowledging the NMS. The system also lacks an explicit method of searching. The problem of finding data within previous systems has not been solved with PeerEnabler.

###

section.5
system.design

5.1 basic.theory

We arrive at the idea of a database that is distributed with replication amongst a network of computers, which are connected using the peer-to-peer model. Based on its simplicity and strength, a logical ring of nodes for the network proved to be optimal. This system plays off the versatility and strength of the peer-to-peer model and allows for an easy method of searching the entire database. Other possibilities for configurations would be a random graph of nodes or perhaps a complete graph. The problems with randomness are fairly clear – it becomes immediately evident that searches would be difficult to perform, there would most likely be issues such as queries hitting the same server twice, loops, etc. The idea of the complete graph is tied in with these problems as well – the question of how to search a complete graph without a central entity and without a high degree of overlap is virtually impossible to answer. That said, the ease of implementing a logical ring along with the ability to search comprehensively and simply made it the optimal choice. Storing data is now scalable, although finding that data is not. Lookup tables, originally from the Chord system, have been completely eliminated.

The system presented here is a database that is distributed (with replication) amongst a logical ring of servers (henceforth referred to simply as the Ring). Using secure[9] peer-to-peer networking, each individual server is connected to the Ring by "knowing" exactly two neighbors[10]. Below is an illustration of the Ring (where the nodes on the Ring are the grey boxes, the thicker black lines connect the Ring, the black dots are clients querying the database, and the thinner black lines connect the client to a node):

[9] Security will be presented later on in this paper. Humor me for now.
[10] By "know," I mean they have information about the server, including IP address, etc.

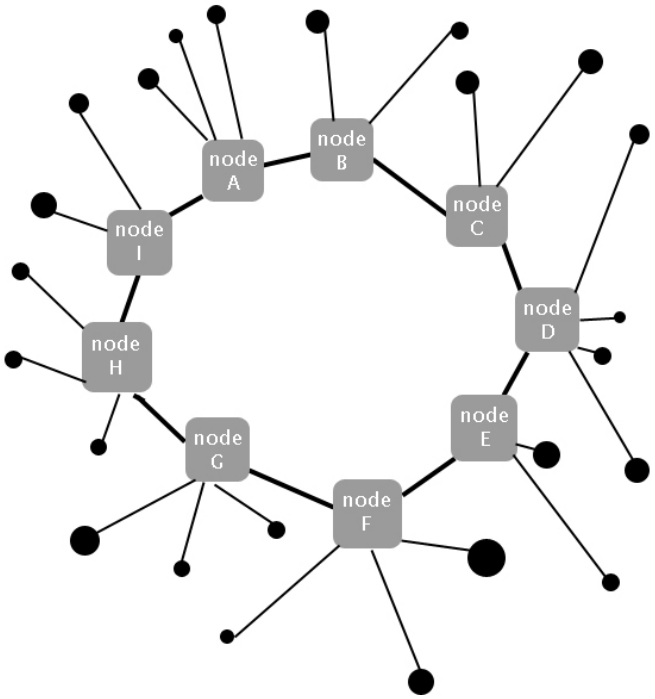

Figure C

The system clearly exploits the recent advances in high-bandwidth connections, processor speeds, etc. At first look, one might assume a distributed database would be advantageous due to the ability to split up the storage space of what would be one database worth of data among many (e pluribus unum and all that jazz). However, this would be a naïve approach. While that thought is true, the principle strengths of this system are its reliability and its flexibility. In the traditional model, if the one server fails, all the data associated with it becomes inaccessible. With this über-system, if one server fails, then you only lose a small fraction of the entire database. As an added bonus, some of its data will most likely be cached in other parts of the Ring. Basically, every time a node receives data from another node, a copy of that data is kept in the requesting node's database. Thus, if a certain piece of data is very popular, it will end up in the databases of several nodes. This feature gives the Ring a predisposition to popular data. Obviously, if a piece of data is cached on several nodes, it is less likely to disappear any time

soon, at least due to node failure. [To allay any fears, the Ring does indeed have the ability to "heal" itself, should a server go down for whatever reason; more on that on the 11 o'clock news. If you cannot wait that long, then please skip ahead to Section 5.2.9.] The increased flexibility comes into play with the fact that each node is responsible for adding, updating, and deleting its own data – gone are the days of one system administrator for an entire database. Once the Ring is setup, it virtually runs itself. While nodes come and go, the Ring remains intact.

The Ring does indeed adhere to the ACID test for databases. Because of its use of MySQL, atomicity is guaranteed – if a part of a transaction fails, then the whole thing will fail. In terms of consistency, the Ring is able to uniquely identify each piece of data on the Ring by marking it with the originating server's name, the author's name, and the date and time of creation (as well as the date and time of the most recent modification to avoid confusing any two versions of a single piece of data). Any two pieces of data can only be confused if they are created by the same person on the same server at the same second. Unless malicious users are members of the Ring, all will be safe. [The members of the Ring will be controlled since a current node must explicitly allow a server to become a node by exchanging public keys with it. Thus, malicious intent would probably not appear in this system.] Isolation is ensured since only the author can update a piece of data. As far as durability goes, data is not routinely mirrored elsewhere. However, popular data will always be on more nodes than not, due to the caching nature of the Ring. This basically means that every time a server requests a piece of data, it will be locally cached there. Thus, if a specific piece of data is extremely popular, that data is essentially mirrored on many nodes.

The Ring provides users with a variety of functionality, most of which should be familiar to database veterans. One can add oneself to the Ring as well as query, update, flush, or delete

data. As you will soon see, these operations provide everything a user could need in terms of distributed databases.

5.2 the.guts,pt.1:the.server

5.2.1 the.beginning

There are two essential components to this system: the Client and the Server. Since a P2P paradigm is used, both are resident on the same machine. The server runs continuously and handles most communications with other nodes on the Ring, while the client is a higher layer that consists of web-based scripts and provides access to the Ring to end-users.

The first thing a computer must do in order to become a node on the Ring is to find a "friend" (a computer that is already a node). We have changed the names of these nodes in order to protect the innocents. Let's call the new computer X and the computer that it knows, A. A's neighbor is B. X asks A to be let in the Ring; A responds to X by telling it information about A & B, as well as telling B information about X. X then is placed in between A & B. The following illustrates this initiation of a node:

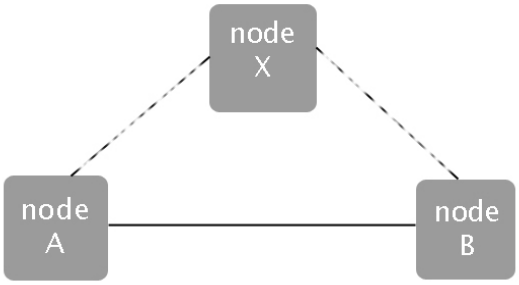

Figure D

The solid line is the original configuration of this part of the Ring. The dashed lines are the Ring after X is included. The solid line connection will disappear. The end result is that X is now neighbors with A & B, and A & B are no longer each other's neighbors. If multiple servers wish

to enter the Ring simultaneously at the same place, then the "friend" of the new nodes will simply add the requests in the order in which they were received; in other words, the matter will be taken care of utilizing atomic serialization.

After setting up this initial configuration, there is a server script that is always running on each node with an open port, waiting for a connection. Once a connection is received, a string of text is sent to the node, with the first portion indicating: query, update, flush, data, stop listening, key, add a node (if you are the A or the B, from the case above), self-healing process (backwards or forwards), or error.

5.2.2 query

A query is a request for information; when a query is received at a node, it queries its local database and sends relevant data directly back to the node from which the query originated (henceforth known as the 'origin'). The query should be forwarded along to the next neighbor of the node. After the end of the query's life (designated by number of results or number of nodes), a "stop listening" command should be send to the origin.

If a query originating from a node is successful, the server on that particular node will handle the incoming data. It will then insert the data into the database. [The query will originate via the client scripts; please see Section 5.3.]

A "stop listening" command is issued once a query is dead. The origin receives this command and tells the client script that all matching data has been received.

5.2.3 update

When an update command is received, the node updates the relevant data in its local database and forwards it along to the nodes to which it has sent copies.[11] Within the local database, each node keeps track of an "update tree" of who it has sent copies to. The update tree consists of a family in terms of the current node; it will keep track of its children and grandchildren, as well as its parent (the node has directly received data from its parents and has directly sent the data to its children, who have sent the data to their children (the current node's grandchildren)).[12] Because of this, an update does not have to travel around the entire Ring. It starts at the server from which the data originated and is sent to those nodes that have received the data from it. From there, each node would forward the update along to the other nodes that have received the data in question from the particular node. If a node receives an update that it has already processed, that update dies (meaning, it is just ignored). This prevents unnecessary loops.

Let us examine a case in point. In the following figure, a piece of data, X, originates at node F; node A requests data X from node F; nodes B & H get data X from node A; and finally, node E gets data X from node B:

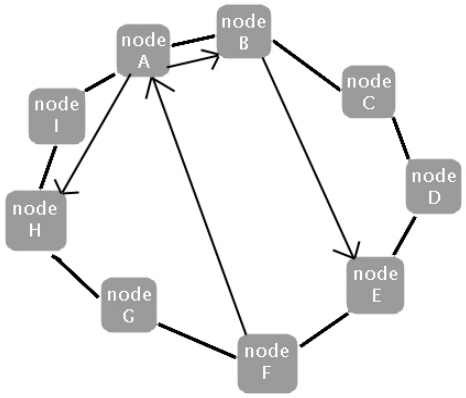

Figure E

[11] Please see Section 5.3.3, the Client Scripts section for adding new data.
[12] The family tree has not been tested in an implementation beyond keeping track of children.

Once the user updates data X at node F, F will forward the update to node A; node A will forward to nodes B & H; and node B will forward the update to node E. Clearly, all the nodes that have data X will receive the update. If a node happens to go down at this point, the Self-Healing Process will take place and order will be restored in the Universe (please see Section 5.2.9 below for more on this topic).

In the case that a parent cannot communicate with a child, the parent will forget about him or her and communicate directly with the grandchildren. In this way, another level of reliability is built in. Otherwise, if a child failed, the outdated data that resided at a grandchild's node would still be floating about the Ring. However, it should be noted that if a parent is having problems communicating with his or her child, then the problem cannot be solved by bypassing the child and talking directly with the grandchildren; this might make matters worse!

If a node decides to be kind to the Ring and "gracefully" leave, then family trees will be updated: grandparents will now take care of grandchildren, since their parents have decided to abandon the project. If the node ever decides to return, then they will not be able to take back their children again. [If a node does not gracefully leave but returns to the Ring later, then all locally cached data will be deleted, in order to prevent obsolete data from floating about.]

5.2.4 flush

A flush occurs when an author wants to delete from the Ring all instances of a piece of data that he or she created. When a node receives a flush command, it will forward the flush to its children. Once this is completed, the node will delete that data from its local database. The timing here is important – if the node were to delete the data first, then it would not know to

whom else to send the command. This would probably result in unwanted copies of data floating around the Ring.

The life of the flush command is similar to that of the update command, as is its use of the family tree.

5.2.5 self.healing.process

The Self-Healing Process occurs when a node on the Ring fails. This will occur whenever a node fails to communicate to a neighbor within a finite period of time. The neighbor is assumed to be dead. Basically, a bridge is formed across the two "broken" ends of the Ring in order to make the Ring whole again. The first step is to go backwards along the Ring until it is no longer possible to keep going.

Once this occurs, the last node sends its information the other way along the Ring until it reaches the node that initiated the SHP. These nodes now have each other's public keys[13] and can replace the dead neighbor with a brand-new, fresh, live one! Thus, we have a whole Ring again, secure and robust!

The situation may arise where two nodes fail simultaneously. When this happens, the SHP will think it has reached the opposite end of the Ring, while in actuality, it is only a fraction of the way there. The result of this near-catastrophe is merely that two Rings will exist independent of each other. While this is an interesting case, it does not cause failure on any large scale.

[13] Security will be presented in Section 5.4

5.2.6 error

An error to the origin merely tells the client script to stop listening. This could happen, for instance, if the SHP has been activated. The end-user will learn about the temporary down time and will be told to try again in a few minutes. At this point in time, the SHP should be complete, and the user will regain happiness.

5.3 the.guts,pt.2:the.client.scripts

All the client scripts are web-based. They include all the basic functionality you expect from a normal database, such as: query, display the local database, as well as inserting, updating, deleting, and flushing data.

5.3.1 query

Without loss of generality, the first thing we should do is query the local database. In doing so, you can save time and energy. However, if the relevant data from the local database is insufficient, then the query is forwarded along to the next neighbor. While the query is forwarded along the Ring, the client script waits for its local server script to tell it to stop waiting. Once this happens, all the relevant data that it has acquired will be displayed on the webpage to the end-user. Also, the data that it has obtained from other nodes will be cached locally, so they can be readily accessible in the future. Only items queried may be cached, but this is logical since only popular information would really need to be cached.

5.3.2 display.local.database

What this command does is left as an exercise to the reader.

5.3.3 insert.data

This merely inserts data to the local database. Even if the data is identical to data preexisting in the database in other ways, one can still add this data – the system relies on time and date created/modified parameters in order to uniquely identify data (in addition to originating server and author).[14] Only if the data is inserted at the same second with all the same parameters will the database become confused. This is highly unlikely as long as malicious users do not exist.

Data is never routinely mirrored onto another node; however, if a node requests data, it is locally cached in that node's database (as explained above in Section 5.3.1).

5.3.4 update

This action updates the local database per the user's request and then forwards it along to the nodes that are known to have copies. The server's job here is explained in Section 5.2.3.

5.3.5 delete

Delete just deletes the data from the local database. This can be dangerous though, as copies of the data could be floating around the Ring.

5.3.6 flush

The natural extension to 'delete' is to flush a piece of data from the entire Ring. This command is similar to 'update,' in that it forwards the command to serves known to have copies.

[14] Clock synchronization is not required here. Since the piece of data can only be updated from the server from which it originated, there will not be problems in terms of inconsistencies among clocks on the Ring. As long as any particular node doesn't suffer from random clock changes, the Ring should remain fully functional.

However, it deletes the data from its local database. The server's job here is explained in Section 5.2.4.

5.4 security:strength.or.weakness?

The one weakness of this system is actually derived from a strength. The Ring relies on PGP's ® public/private key security to encrypt, decrypt, sign, and authenticate various packets that are flying around the Ring.[15] However, this increased security slows down the system quite a bit. In designing this system, I had to weigh the cost and benefits of faster speeds and less security versus slower speeds but more security.

Public and private key encryption is one of the more secure forms of cryptography. In using this method, one would keep his or her private key (which is secured with a passphrase and an actual file on the user's computer) and send the public key to anyone who may wish to securely communicate with the user. If Dr. Hearn[16] wants to send a secure email to Dr. Anderson,[17] he would first obtain Dr. Anderson's public key and then communicate in another fashion, such as by telephone or in person, to confirm that the public key had not been tampered with in transmission. Dr. Hearn would encrypt the email with Dr. Anderson's public key and then send the email. Once received, only Dr. Anderson's private key (along with his passphrase) would be able to decrypt the email. Thus, communications can be transmitted securely over relatively open lines without fear of someone being able to read said communiqué. This method of security can also be used to sign a piece of data, as well as authenticate it on the other end.

[15] MIT Distribution Site for PGP
[16] President of Wake Forest University
[17] Vice President for Finance & Administration at Wake Forest University

From the originating server to the last node during a query's life, each neighbor authenticates its previous neighbor's signature and signs the query with its own. The originating node's public key is passed along in this manner, so whenever data is to be sent, the server with the data can know with confidence the authenticity of that key. If the origin does not have the latter server's public key,[18] then it should send it as well. However, in order to maintain the integrity of the security of the Ring, the key needs to be passed along the Ring (via its neighbors), while the data can be sent via a direct connection between the origin and this node. This creates a ring of trust, where each node trusts its neighbors, who trust their neighbors, and so on, ad infinitum. Thus, each node inherently trusts all other nodes on the Ring, as long as communications travel via the neighbor system. Without loss of generality, this security clearly prevents any sort of "Man-in-the-Middle" attack. When a node sends a piece of data across the Ring (in other words, to a non-neighboring node), that data is encrypted such that if someone else were to tamper with it, then the receiving node would merely discard the data. Also, the man in the middle would not even be able to read said data, since it is encrypted using public/private key security.

As far as data transmissions go, the server will check to see if it has the node's public key. If not, then it will open a new port (so as not to conflict with the main port) and wait for the key. Once this part is complete, the node authenticates and decrypts the data, and inserts it into the database.

In order to add a server into the Ring, the new node and its friend must have communicated previously and exchanged public keys.

The whole purpose of the SHP is that the Ring can become whole again. It does this securely. By only communicating with neighbors, the SHP is somewhat unscalable on a large-

[18] A table is kept on each node with a list of which servers have the current node's public key.

scale Ring. However, this is inevitable with the current system. The elegance of the SHP lies in its simplicity and effectiveness. By corresponding via the Ring of trust, communication is always secure.

###

section.6
implementation

The preceding design was implemented using the PHP: Hypertext Processing scripting language and the MySQL Open Source Database; all programs were written using PHP, and the database was used for storing data and information about data, as well as information on Public Keys. "PHP is an HTML-embedded scripting language. Much of its syntax is borrowed from C, Java and Perl with a couple of unique PHP-specific features thrown in. The goal of the language is to allow web developers to write dynamically generated pages quickly."[19] PHP was an optimal choice for its simple, yet effective, means of networking, as well as its more common utility as a web-based scripting language. While I was able to find some instances of PHP being used in a server-client environment, little is available concerning peer-to-peer implementations using PHP. Additionally, sockets are a relatively new addition to PHP, so there is not as much extensive documentation on them as might be in another language. "The MySQL® software delivers a very fast, multi-threaded, multi-user, and robust SQL database server. MySQL Server is intended for mission-critical, heavy-load production systems as well as for embedding into mass-deployed software."[20] MySQL was an obvious choice both for its reliability and its cheapness – being open-source software, MySQL and PHP were excellent choices in terms of cost and trustworthiness.

Within the database are two principal tables in which all information pertaining to the Ring resides. The primary table contains in each entry: the author's name, his or her email, the date and time created, the most recent date and time modified, the name of the originating server, and locations of nodes to which the current server has sent this piece of data. Within this

[19] Php.net
[20] Mysql.com

particular implementation, the database will also contain the radio affiliate, the topic of the review, the genre, the website for the artist, the label, contact information, and the review. In its place in other implementations would be the piece of data and/or the location of that data on the server (depending on what type of data it is). The other table in the database contains information on which nodes the server has sent its public key to.

On the simplest level, the main server script starts listening on a port. When it accepts a connection, a large switch statement handles which activity the requesting node wants. First, we will take an in-depth look at the query. This command comes in with the following information: the name of the originating server, the query command, a result count, and a hop count. Either the result count or the hop count will be set to 0, meaning to ignore it, and the other will be a value that is decreased as the query travels around the Ring. By allowing for a hop count, one can know exactly how many servers one has hit and can have a general idea of how many servers to hit in the future. The server will check the database to determine whether the query returns any matching data that the requesting server does not yet have. If such data exists, then the server will first update its database to keep track of where the data has been sent. Afterwards, the data will be sent directly to the originating server. If either of the count variables have finished their run, then the query is effectively dead, and a command is then sent to the originating server to let it know of this fact. Otherwise, the server forwards the request along to its neighbor.

The next command we shall look at is update. The first thing the server does is to check if the update has already been applied to itself. If the date and time at which the piece of data was last modified is the same, then the server knows that it has already processed this update, and can kill it immediately. Else, it will perform the update and forward the update to each

server to which it has sent the piece of data. The flush command is almost identical in form to the update, with the exception that instead of updating the data, it will completely delete it from the Ring.

In order to develop this system while simultaneously testing and debugging it, four computers were used in a small LAN. Within this limited environment, all features described in the previous sections were fully operational.

###

section.7
conclusions

While progress has been made in the realm of database technology, most advances remain unscalable, making them inconsequential in this era of bigger, better, faster. What programs such as Kazaa give to the community is an enormous distributed and scalable database for the explicit purpose of file-sharing. The system presented heretofore provides something similar, yet more comprehensive and inherently more secure. In most P2P systems, searches follow strange, unpredictable paths via a rather irregular graph of nodes. Security in these systems is just about nil.

The system proposed in this thesis provides a complete database solution based in MySQL that is distributed with replication, secure, comprehensive, and scalable. By organizing the P2P nodes in a logical ring, searches are predictable and have the option of being complete (meaning they could search the entire network). Security is ensured using the popular method of public and private key encryption.

In terms of scalability, we can examine it in terms of both space and speed. The former refers to the space needed on each node in order to maintain the Ring – this overhead will include the space of the scripts, public keys of various nodes on the Ring, and the local database. The first two are minimal, while the local database can grow. However, the administrator of the local database can determine how often to get rid of old, unpopular, cached data. With respect to speed, the system is scalable up to a medium-sized system. Due to the security measures in place, the Ring cannot compete with such large-scale systems as popular file-sharing programs; however, these systems are typically not secure.

To return to the ACID Test for databases, we evaluated this system for its atomicity, its consistency, its isolation, and its durability. Since MySQL provides the essential framework for this system, atomicity and isolation can be guaranteed (if one part of a transaction fails, then the whole transaction fails, and transactions will be locked out of a local database while another transaction is occurring). On a network level, isolation is ensured because only the original author can update a piece of data. If someone else's data has been cached on a machine, it cannot be updated unless a superuser physically enters the MySQL database to change it. Consistency is ensured with unique identifiers; namely, every time someone creates a new piece of data, it is marked with the originating server's name, the author's name, and the date and time of creation (if it is modified in the future, a separate date and time stamp is updated as well). Thus, the only way two pieces of data can be confused is if they are identical and were created by the same author at the same second of the same minute of the same day. If this is happening, then we have more things to worry about than consistency (such as reevaluating Einstein's notion of simultaneity). Durability is an interesting topic – while data on servers is never routinely mirrored or backed up, popular data will inevitably be cached on other machines. Additionally, if one machine goes down, you still have the rest of the Ring that you can rely on for data. Thus, the Ring can stand up against the ACID Test and be known as a viable database alternative.

While this system has the ability to stand on its own, there are areas of research that can be explored. I have relied on the ability to exchange public keys prior to adding a node to the Ring. I originally desired to have a system that was completely devoid of any central authority,

even a registry of keys elsewhere on the Internet. However, one could view a global registry of this sort as a public Internet service that could add more to this system than detract from it.

Due to the limited scope of this project and problems incurred in attempting to apply security to the entire system, the following commands are not secure: stop listening, updates, and flush. These areas could be easily patched without too much more research and development.

A bit of added functionality could be the ability to "search more," where if a search did not return the desired results, the search would continue from where it left off.

Something interesting to look into would be caches based on probability. While each node currently caches all incoming data that it has queried, another option would be to attach a sort of popularity rating to each piece of data and only cache data that has a certain degree of popularity.

To increase scalability, one could implement the system in terms of multiple Rings. Within this specific case, the Rings could be based on genres. One option would be that each node could join several different "genre-Rings" (e.g., Rock, Jazz, Hip-Hop, etc.). Another option would be that there are several Rings that are interconnected using a backbone Ring, which would allow you to query multiple Rings in parallel.

An intriguing research problem would be to implement this system on a wireless network. An example might be if you want to manage data on small handheld devices. Clearly, any one device cannot manage an entire database that services many others. Thus, the Ring would be a good choice to use on this sort of network. However, the situation may arise where a node can only communicate with one other node (thus making it impossible to have two distinct neighbors on the Ring). When this happens, there should be no problem in having one node as both of your neighbors.

Another interesting use of this system would be to apply it to online blogs. Case in point, several blogs exists for the purpose of providing technical support for various Linux distributions. Each blog would be connected as nodes on the Ring – one could query the Ring for general Linux support and receive answers to questions posted on several different "distro" blogs.

While there are several directions in which this system can go, the Ring as presented within the contents of this paper is a robust, secure P2P database that is distributed with replication and needs no central authority to maintain its integrity. The Ring and I appreciate your time and consideration.

###

section.8
bibliography

Gnutella.com. http://www.gnutella.com/. 6 April 2004.
Joltid. http://www.joltid.com/. 6 April 2004.
Kazaa.com. http://www.kazaa.com/. 6 April 2004.
MIT Distribution Site for PGP. http://web.mit.edu/network/pgp.html. 6 April 2004.
Morpheus.com. http://www.morpheus.com/. 6 April 2004.
MySQL Online Documentation. http://www.mysql.com/. 31 March 2004.
Napster.com. http://www.napster.com/. 6 April 2004.
Parker, Jason. "The ACID Test - A mini database tutorial for those that are interested." ArchWing. http://www.archwing.com/technet/technet_ACID.html. 31 March 2004.
PHP Online Documentation. http://www.php.net/. 31 March 2004.
Tanenbaum, Andrew S. Computer Networks. 4th Ed. Prentice Hall PTR: Upper Saddle River. p380-384.

section.9
acknowledgements

Dr. Errin Fulp
Adam Bregenzer

section.10
appendix

The remaining pages consist of the code developed during the Summer of 2003 in researching this Thesis.