WAT: A Distributed File Sharing System

Global knowledge of widespread storage

6.033 Design Project II

Recitation: Snoeren/Friedman TR 12

Wolfgang Gatterbauer, Aaron B Strauss, Toh Ne Win

{wolf, aaronbs, tohn}@mit.edu

May 10, 2001

ABSTRACT

WAT is a distributed peer-to-peer file sharing system that uses a level of indirection to separate the storage of files from information sources about storage. It uses well-defined mathematical operations on a file's unambiguous name to determine such globally known information sources called primary servers. For any given file, such servers are organized in a demand-adaptive tree so information can be efficiently replicated. Actual files are with the users who download them. The use of an asymmetric key system allows users to verify integrity of files and only authors to update them on the system. By relying on randomization and secure hashing and by operating on a "trust nobody" basis, WAT can prevent impersonation and denial of service attacks efficiently. The time for file lookups increases logarithmically with the total number of nodes on WAT, while the replication capability is increases significantly.

1. INTRODUCTION

The Internet, through DNS and HTTP, has facilitated accessing public materials and data very well. However, many users do not have the resources to post information and software for others to download. Publishers may lack space to hold their documents, or more likely, sufficient bandwidth to meet users' demands. Therefore, we set out to design system that allows publishing and high demand downloading of documents on a decentralized network of independent computers.

WAT is designed as a decentralized system in order to utilize the additive effect of thousands (or millions) of users on disk space and bandwidth. Centralized systems, such as Napster's music sharing network, disallow a user from posting data that he/she cannot store on his/her local node. These networks also require that centralized servers be working, and therefore low availability can be common in these systems.

Decentralized systems, on the other hand, are more robust. If nodes fail one at a time, each node can independently assess the state of the network and replicate files as needed. Under this situation, a decentralized network should be able to successfully entertain all requests as long as there is enough total disk space to handle all the files. If nodes fail (approximately) simultaneously then decentralized systems might not be able to locate all theoretically available files.

The entrance to the WAT system is hyperlinks on publishers' web sites. Each link has a document's public key as meta data, which direct users to servers on the WAT system. These servers do not directly provide data, but connect users to other users' nodes that previously downloaded the requested document. By including this level of indirection, WAT gives users more authority to manage the space they donate to WAT, and straightforwardly introduces a demand-based feedback loop.

2. PREVIOUS WORK

Two systems that are addressing the decentralized file sharing problem are Gnutella and FreeNet. Gnutella passes file requests from one node to another. This node hopping provides anonymity that we do not seek to implement. Instead, we focus on the user being able to retrieve a file on our network. Even with all servers running, Gnutella can not guarantee a user will find the file that she seeks: the number of servers between the file and the user might be greater than the maximum hops allowed.

Gnutella is often unreliable because it is so susceptible to denial of service attacks. Since all nodes in a user's *horizon* (i.e., inside the hops-to-live value) are involved in the search for one file, service slows when users initiate several frivolous requests or use Gnutella to chat.

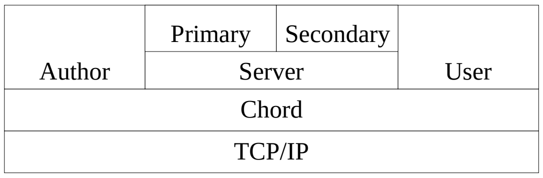
FreeNet has many of the same goals we have in our design. Publishers using FreeNet are able to modify their existing files, a feature included in WAT. However, there are two main drawbacks with FreeNet. One, FreeNet does not make any guarantee about how long files will remain on the network. Publishers' documents should not disappear within a few hours just because the newest Britney Spears video was released and demand was too high to accommodate the new files. Two, FreeNet—which implements a node-hopping scheme similar to Gnutella—is structured in such a way that searching time is linear to the number of nodes on the network.

Even though it is centralized, Napster introduced several important ideas in the peer-to-peer arena. First, users control what files are stored on their computers. Second, since files that users download are public, the more a popular file is downloaded, the more servers exist from which another user can download this file. After a popular file experiences a wane in demand, multitudinous copies of the file still exist, but a user can delete the file from her system if her disk space is limited.

3. DESCRIPTION OF DESIGN

3.1 Layers of WAT

The WAT system runs under one software package that consists of three modules: the author, user and server programs. All three run on top of the Chord system [4], a distributed hashing technique that maps 160-bit keys to IP addresses of hosts on the system. Chord itself runs on TCP/IP. The basic unit of storage in WAT is the *document*, which has a unique name that stays constant through version updates.



3.2 Document Naming

When an author wants to submit a document, he generates 1024-bit RSA key pairs K_s and K_p. He keeps the secret key K_s and uses the public key K_p as the official name of the document. To publish a document, the author puts K_p on his web site and stores the file, signed by K_s, in WAT. He also provides a time stamp of his clock to the server and on his web site, both of which are included when making the signature.

After a user downloads a file, WAT verifies that the file came from the author by authenticating the signature with the file and K_p (which the user specified). This method ensures that (1) all documents can be checked so that malicious servers cannot send bad data, and (2) all document names are unique.

The data that is transferred to a server at upload is:

Name = K_p	Time stamp	n = # in the tree	Document data	Signature

The naming scheme was inspired by FreeNet [3].

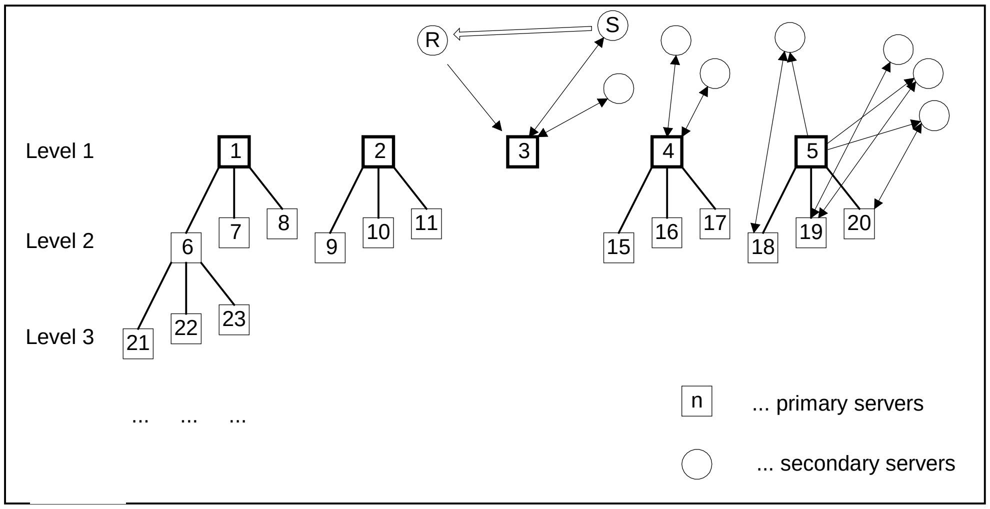
3.3 Primary Server Trees

WAT separates the storage of files from the location that stores information about that storage. Storage is ultimately demand-driven, while the information is stored more deterministically. The system uses Chord [4] as decentralized lookup service to determine the servers that store information relevant for a particular file.

For adaptability reasons that will become clear in later paragraphs, the primary servers for a file are organized in a tree-structure with branching factor 3 and 5 top nodes instead of a single root. SHA[Kp +

n] gives a 160-bit index that can be mapped to a server using Chord. The Chord ID of node n within a file's tree-space [Kp, n] is the 160-bit SHA-1 hash of the file's key concatenated with the number n:

`Chord-ID(`K_p`, n) = successor(SHA-1[`K_p` + n])`.



A node's parent is node (n-3) div 3 and its children are {3n+3, 3n+4, 3n+5}.

This design enables global knowledge of the nodes of each file's tree-space within the network by using the interface lookup(SHA-1[Kp + n]) of Chord. Similarly, a server can check if it is supposed to be a primary server for a document if n and key are supplied.

3.4 Document Insertion

The author produces a random public/private key pair. He adds a timestamp and a signature to his file and uses the command `insert()` to send signed requests to 5 top *primary storage servers* (PPSs) to allow him to upload his file. Upon receipt, a PSS checks the following:

- Looks into its storage for a version of the file and only proceeds if the file does not exist or it exists and its timestamp is older.
- Verifies the signature with the key and timestamp.
- Verifies that key + its advised node number maps to itself in the chord system.

If all nodes accept, the file is transferred. If one node does not, `insert()` returns an error message. In such a case, the author can generate another random key pair for another set of servers.

3.5 Document Replication

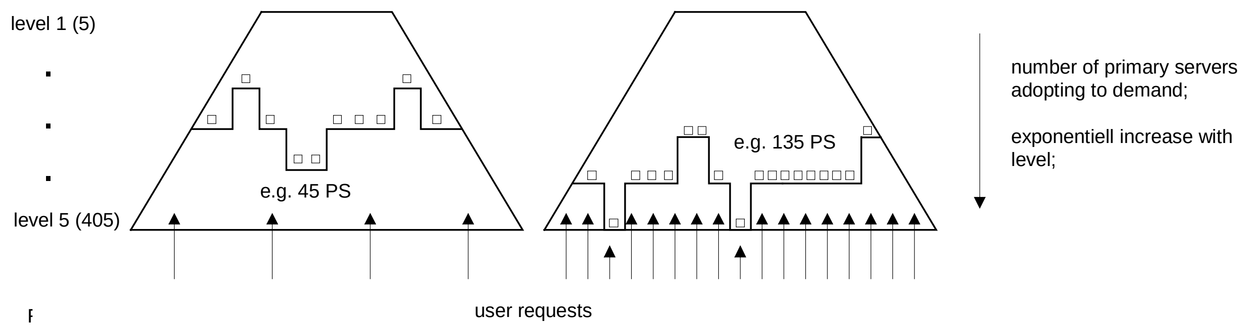
3.5.1. Primary versus Secondary Servers

PSSs have the dual role of storing files and pointers to other storage. The first users will download a file directly from a PSS. The user then informs the PSS via the `reportSS` that she has the document and becomes a *secondary server*, a server that provides storage only. The PSS adds her computer's IP to its pointer cache list. From then on, the PSS can act as a *primary server* (PS)—when a new user requests the same file, a PS simply directs the user to a secondary servers it knows about.

By default, the author and any other servers he designates become the first secondary servers of the system. This is so that a large publisher can provide hosting space for its documents to satisfy normal demand, but still have WAT's capability to serve data under high demand. An author can chose not to become secondary server as well.

An SS keeps a pointer to the PS while she has the file in her WAT storage system. She informs this PS when she starts up, goes down or purges the file and also declares her bandwidth with the `reportSS` message (this is later used in managing storage). A secondary server remembers only one "assigned" primary server, which might change over time, as later explained, but several primary servers might point to one secondary server.

When a primary server's pointer cache exceeds ten entries (a somewhat arbitrary value) it stops acting as a storage server (i.e., allowing document downloads) and becomes a pure pointer server. This transformation means that when demand for a file goes up, primary servers become responsible only as directories and are not heavily loaded. For popular files, bandwidth for downloads is shared between secondary servers while contact with primary servers is kept to small messages.


level 1 (5)
level 5 (405)
e.g. 45 PS
e.g. 135 PS
number of primary servers adopting to demand;
exponentiell increase with level;
user requests

3.5.2. Tree Replication

When demand increases more, primary servers may become loaded by pointer requests. The tree structure alleviates this demand by allowing primary servers to replicate when either:

- The busyness counter associated with a document exceeds a threshold
- The size of the pointer cache exceeds 90% of the capacity for the document

A sever replicates to its children by copying the document's pointer cache entries via the `replicate` command. Cache entries are divided equally between children. Furthermore, a server replicates only once to a child; if a child already is on the tree as a primary server it simply refuses the replication request.

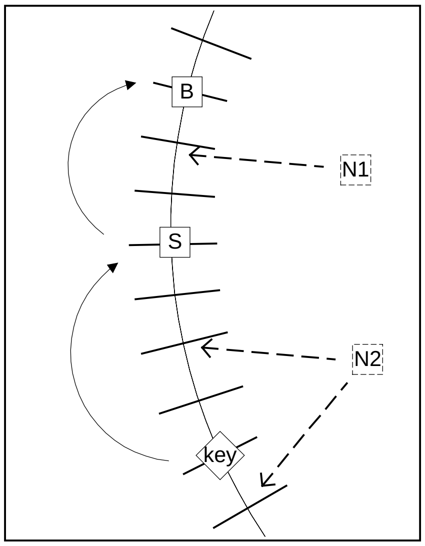
When a child receives a `replicate` request, it checks to be sure that it was from its parent by using the tree structure. On a legitimate replication, the child becomes a primary server for the document (without storage) and starts to accumulate its own list of secondary servers (these new servers are not told to the parent). The child also informs the secondary servers it received that it is now the primary server for them, via a `pointing-to` message. The child can replicate to its children if the above conditions also apply to it, with a maximum tree depth of 5 levels.

Combined with deletion described in the next sections, tree replication ensures the following (within the context of a document):

- If a node is a primary server, its top-level parent nodes are very likely to be primary servers.
- Since users do lookup from the bottom up (see next section) the amount of load at each node will never become excessive—nodes will replicate to their children and subsequent users will contact their children first. Allowing 5 levels provides for 405 primary servers at level 5 if necessary and 605 servers total.
- Multiple primary servers may initially point to a secondary server, but each secondary server will point only to the lowest primary server that points to it (because of `pointing-to` messages).

Since Chord and SHA produce random hashes, two nodes along a document's tree might map to the same server. In this case, a primary server simply keeps its old position when a replication request arrives.

3.5.3. The "Buddy System"



Over time, primary servers will start up and go down. In order not to lose information, WAT keeps a redundant record of primary servers. A PS regularly mirrors its information to its buddy, which is its successor in the Chord-system. Chord permits WAT to know about changes in the Chord Topology through its API. A server and its buddy can therefore know if they get replaced in the tree structure of a certain file.

Crashing or voluntarily leaving servers means the same to WAT: this distinction is hidden below the abstraction of Chord's API. If a server leaves, its buddy is notified and transparently replaces the first server. If the server does not have time to mirror its most recent pointers, some data might get lost, but this only marginally affects WAT.

When a new node starts up, it will call `join()` to be integrated in the Chord system. It will then be ignorant of its tasks as a server, but `event_register(fn)` will notify the new node's successor and predecessor of its presence, and will decide whether to transfer new tasks to the new node. In the figure on the right, N1 would learn through B to be the new buddy of S. Server S—depending on N2's relative position to the key that maps to S—might decide to copy its information to N2.

3.6 Document Lookup

Once a user has a document name from an author's web site, she can locally determine which servers are acting as primary server for the document using the tree formula. Our protocol requires her to use a bottom-up search. She picks a random node at the bottom of a document's tree and traverses up until she finds a node that is already a primary server, not busy, and willing to supply the document. In most cases, she will find a primary server that is not busy (see analysis for other cases) using the `getSS` message. Appendix B details the exact algorithm.

Once a good primary server is found, it gives the user a list of ten secondary servers it believes have the document. The user can then ask each secondary server for the document until she can `get` it. For each secondary server that doesn't have the document, she sends a `report-SS` message telling the primary server to erase that server from its pointer cache. If all ten do not have the document, the user restarts the primary server search on another branch of the tree at the bottom. As siblings do not interact with each other, no malicious primary server can prevent users from getting information after several attempts.

From the perspective of a primary server, there are three main actions involved upon user requests. First, if it is too busy to serve pointer requests (which should only happen in rare circumstances) then it simply does not respond to the user (so there is no extra "busy" message). Second, if it is asked for information that it cannot supply (usually, because the parent did not transfer it), it sends a NACK. Third, if it is able to send a list of ten secondary servers, it chooses them in a round-robin way. Hence a primary server will send entries 1-10 to the first user, 11-20 to the next one, etc. This insures that each secondary server is equally used.

3.7 Managing Storage and Deletion

Of course, no primary server can keep pointer caches forever and no secondary server will keep documents forever. Each type of server follows rules based partly on user feedback so that WAT dynamically heals as problems arise and adapts as demand changes.

3.7.1. Primary Servers and Pointer Caches

The five top primary servers are required to keep files for at least one week, which is the minimum period we guarantee that a file is available on WAT. After this time, these top servers expunge the file from storage, but remain primary servers by keeping pointer caches.

Ordinary primary servers maintain their pointer caches using a chained hash table (see Appendix A) ordered on recent use. When the server runs out of space (because of `insert` or `replicate`), it expunges the least recently used (lowest demand) pointer cache. When a cache is expunged, its pointers are copied to the server's parent (via the `purging` command), who keeps the secondary servers informed about the change. This insures that if a document loses popularity, the top-level primary servers will be the last ones to hold pointer caches for it. Otherwise, if only a few bottom level servers held pointer caches, users may not be able to find them with a randomization algorithm.

Lastly, primary servers have to manage inside each individual document's pointer cache. Secondary servers in the cache are assigned a priority based on declared bandwidth. When a new secondary server declares itself and there is no more space in the cache, the pointer with the lowest bandwidth is expunged and the new secondary server is added.

3.7.2. Secondary Servers

A WAT user designates an amount of space to be the secondary server cache (see space requirements). When a user downloads a document, a copy goes into the secondary cache (the other copy is sent to wherever the user wanted). With this double copy method, users can manipulate downloaded files and WAT can manage the cache as a black box.

As expected, the secondary server cache also fills up as documents arrive. Secondary cache entries are stored using chained hash tables ordered on last access time (by other users wanting the files). If no document has been recently accessed, the user's access times are used instead. When space runs out, the last accessed document is expunged and the appropriate primary server is informed.

WAT also allows for secondary servers going on and off intermittently (e.g. when users shut off machines at night). When a secondary server shuts down, it tells the relevant primary servers (with `Report-SS`) so they may update pointer caches. When it comes back online, it tells its primary servers that it is again available. If a secondary server goes down without informing its primary server, user feedback will eventually delete the right pointers, just more slowly.

3.7.3. Deletion and Consistency

We intentionally chose the "replicate once" rule between primary servers to limit communication between them. This has many advantages:

- Instead of having to maintain consistent pointer caches across the entire tree, each server manages its own cache without having to inform the others.
- If a secondary server goes down or comes back up, the SS only needs to inform one node. If a node has incorrect information about a secondary server, it cannot pollute another node with this information. The first node will eventually receive correct information from user feedback.
- Malicious primary servers cannot alter the pointer caches of nodes that are already primary servers (and for nodes that aren't, they will correct themselves from user feedback).

3.8 Updates

In order to update a document, the author has to keep the document's private key. He can then send a signed request with timestamp to all of the 5 top primary servers with the same command as originally: `insert()`. The servers check the signed timestamp and if all accept, the document will be transferred. Upon update, these primary servers will (1) tell all cached secondary servers to expunge their local cached copies, and (2) expunge their pointer caches themselves.

Afterwards the primary servers send the signed message from the author further to the children which repeat the scheme until level 5 is reached and all 605 possible primary servers have been contacted and expunged possibly stored data. From now on, the new version can be treated as a new file and the replication scheme described above starts over.

After the erase, there might still be secondary servers with old data that were shut down during the action. At startup, they contact their assigned primary servers and learn about erasing the files.

3.9 Installing the WAT System

To join the WAT system, a computer simply needs to run the WAT software package and manually type in the IP address of at least one computer already on the network. The computer will ask this specified other WAT system to:

- Register as a Chord node—this is handled entirely by Chord's API.
- Join WAT as a primary server using Chord and the buddy/tree system.

We have claimed in our introduction that we will build a fully decentralized computer system. The seemingly awkward trouble of manually typing in an IP-address is the only way to achieve our goal of complete decentralization. IP addresses could be published on a web page or obtained friends. The WAT software itself would of course have to be downloaded the traditional way.

4. Theoretical Analysis

4.1 Design Philosophy

We strived to design a simple and unambiguous system that fairly allocates resources, adapts to varying demand, and yet guarantees a week of storage for documents. We do not give strong guarantees, because we perceive guarantees as obstruction to overall good performance; we only commit to a "best-effort".

For example, WAT does not give the guarantee that a user retrieves the latest version of a file. It is very likely, but not certain. In the updating scheme, an uncooperative child could refuse to update a file. Although the parent would continue its attempt all the nodes further down in the tree may not get updated.

If a user wants increase certainty about having the latest version of a document (e.g. war news from a secret reporter who posts information at irregular times) and there is no possibility of information exchange about the version outside of our system (e.g. weather info would contain a date), a user can perform a retrieval of a document for several times, and then compare the included timestamp.

Hence, if certain users or authors have special needs that deviate from the majority, such a feature is best implemented at a higher-level, in accordance with the end-to-end-argument[3]. We concentrate on overall high availability and only guarantee that a posted document was from the author that had the private key.

4.2 Space Requirements

A WAT server is required to provide at least a fixed amount of storage to act as a server. The user can choose to increase the space allocated for this as his storage capability grows. The initial requirements are 10 MB for pointer caches (1000 files * 2000 entries * 5 bytes/entry), 100 MB for primary storage and 100 MB for secondary storage. Each server by default can be primary server for 1000 documents with 2000 secondary servers each. Note that uploads are not limited to 100 MB: an author can still choose to ask a server to be a primary server (i.e., without storage) and instead provide storage on his own. Of course, none of these requirements can be totally enforced, but since these are not security issues, we expect that having most users comply with them will allow WAT to work.

If every server on the system obeys the above storage requirements, we can make a few calculations. Given an average file size of 1 MB and 10,000 servers:

- If primary storage is limiting, WAT can store 100,000 files (10,000 * 100 MB / (5*2*1MB)). The 5 is for the top 5 servers and the 2 is for the buddy system.
- If pointer cache space is limiting, there can be a maximum of 250,000 files if each file has 20 primary servers (that's level 1 and 2). If every file is on the maximum 605 servers, this goes down to 8260 files, but this implies that all files would be under heavy demand.
- A full tree for a file has 605 primary servers, so it can accommodate 1.2 million secondary servers (less with some overlap). This should be enough for all files, since 1.2 million replicas are unlikely.

4.3 Performance

By using Chord and a deterministic tree structure, WAT can provide fast access to data for its users. We split up the search for a document into three steps.

- Finding primary server. After getting the file name, a WAT user can quickly know which servers to request pointers, and in the general case, servers will not be busy. The main cost to this is hence 5 calls to Chord, which runs in log of the number of servers. Hence, a primary server can be found in 5*lg(N) communications between computers, and a user can know in 25*lg(N) communications that a file is not present (contact all level 1 servers). For example, with 1,000,000 servers and 100msec per communication (two TCP packets), lookup would take about 10 seconds. This time would go down with popular files since searches would likely end before reaching top-level nodes.
- Selecting and querying secondary servers. If any of the servers work, the maximum time before download begins is another 10 communications. If none of the servers work, then primary server lookup has to be performed again, but this is highly unlikely. In any case, these ten servers will not be given to the next user.
- Downloading from secondary server. The speed of an individual download is limited by the bandwidth between a user and a secondary server. However, by using SHA, Chord, tree branching and round-robin scheduling, we know that secondary servers will be selected randomly from the pool of available ones, loads will be equally spread out between available servers.

4.4 Attacks and Weaknesses

The main principle of operation for WAT is to not trust any part of the system (server, author or user) since any computer may act in any role. We cannot rely on the WAT code staying unchanged on the machines of users. In this section we describe the possible attacks and weaknesses and how WAT combats them.

4.4.1. Security Attacks

WAT allows authentication of authors and prevents malicious destruction of data by providing:

- Authenticity of documents: updates and downloads are checked by signatures. This prevents "fake authors" and malicious servers from uploading bad data or deleting existing data. If authors fear their computer will be broken into, they can throw away their secret keys.
- Authenticity of servers: servers know if they are supposed to be primary servers by checking with Chord. They also know if they should acknowledge a replication request or not. Also, since replications are done only once, bad data cannot propagate to established servers.
- Randomness of servers: primary servers are determined by SHA, Chord and the tree protocol. A server cannot intentionally choose to serve (and disrupt) a particular document.

- Restrictions on uploads and requests: servers do not allow users or authors to make arbitrary requests for downloads and uploads. A server may only allow one file submission from an IP address, for example. This prevents denial of service attacks through excess submissions or requests.

4.4.2. Availability

WAT provides availability of documents with:

- Randomness: with random servers and the tree protocol, requests are balanced and servers will not get busy due to one document.
- Self-repair: primary and secondary servers store pointers to each other, but these pointers change based on user and server feedback.
- Buddy-servers: if a primary server goes down, its buddy can take over. The only time this may fail is if a server and its buddy crash at the same time, but then the tree can be used to reinstate servers.

However, there is one weakness to WAT: the lack of locality. With many levels of indirection to different places on the Internet, WAT may generate unnecessary messages between distant servers.

4.5 Enhancements and Alternate Designs

Here we discuss alternate approaches we passed by during WAT's design.

We considered always throwing away the author's private key to enhance security. This would prevent alteration of files by breaking into an author's computer. However, without private keys, authors cannot modify posted files without changing their names. WAT's final solution is to allow authors to modify existing documents under the same name and push the responsibility of keeping the private keys secret to the author. However, the author can throw away his key if he wants to sacrifice updates for security.

When first designing how to replicate file information, we considered using a linear (non tree) way to find the servers that received replicated information. For instance, if server A was busy with Britney Spears requests, it would replicate Britney's file information to A's Chord successor. As each server down the chain became busy, it would pass on the information to the next node. The current tree configuration is superior because primary servers grow exponentially, ensuring they do not get busy during high demand.

The most important design decision was to (mostly) keep pointers at primary servers instead of the files themselves. At first glance, it seems easier to hold the files in the primary servers since it is the most direct way to access files. However, complex probabilistic formulas would have to be calculated to figure out exactly when to replicate or purge files. Whether to replicate or purge would have to depend on the demand for the file, the space left on the node, the available bandwidth of the node, whether the node's parents and/or children already have the file, etc. By introducing an extra level of indirection into WAT with secondary servers, we simplify this process and take advantage of the user feedback loop.

4.5.1. Empirical Optimization

Our design leaves ample space for empirical optimization. Changeable parameters include:

- Branching factor of primary server trees. We could have 10 level 1 servers with 5 children each, for example.
- Maximum depth of tree. Instead of limiting it to 5 levels, we could allow 10 levels, which would provide 147,000 primary servers per file maximum.

All these parameters would be best optimized through simulation or empirical testing. Since WAT is currently a theoretical design, we did run any simulations for the system. With simulations, we could choose more optimal parameters and ensure that our availability assertions hold. If WAT is implemented, we recommend running these simulations before using WAT for real storage.

5. CONCLUSION

By combining the fast algorithms of Chord, the decentralization of Gnutella and FreeNet, and the file storage principles on Napster, WAT is a reliable, user-friendly, peer-to-peer file sharing system. WAT is shielded from most malicious attacks by implicitly not trusting users, authors or servers. Users who want control over their space and publishers who wish to provide easy access to their data can use WAT and be sure that documents will be available under changing demand.

6. REFERENCES

[1] JH Saltzer, MF Kasshoek. Topics in the Engineering of Computer Systems. MIT 6.033 class notes, draft release 1.10 (6 Feb 2001). Chapter 4.

[2] RK Ahuja, TL Magnanti, JB Orlin. Network Flows. Prentice Hall (1993). Upper Saddle River, NJ.

[3] JH Saltzer, DP Reed, DD Clark. End-to-End Arguments in System Design. IEEE: 2nd International Conference on Distributed Systems (April 8-10 1981). Paris, France.

[4] I Soica, R Morris, D Karger MF Kaashoek, H Balakrishnan, Chord: a Scalable Peer-to-Peer Lookup Service for Internet Applications. MIT, http://pdos.lcs.mit.edu/~kaashoek/chord.ps, May 2001.

[5] FreeNet: http://freenet.sourceforge.net/

[6] Gnutella: http:// http://gnutella.wego.com/

[7] Napster: http://www.napster.com/

Appendix A: Chained hash table data structure

Many of the storage systems used in WAT employ the chained hash table data structure. This section describes its operation in greater detail.

Chained hash tables in WAT are indexed by name and linked in order of last access time. This allows constant time insertion and deletion of data, providing that we insert in increasing order of time (we can delete arbitrarily). To create a chained hash table, we set up a normal hash table with the name as key. Each entry in the hash table, in addition to storing data, has two pointers: prev and next, which point to other entries to create an ordered, doubly linked list on last access time.

When any record is removed, its prev and next records are appropriately updated to keep the linked list in proper order. When a new record is inserted, it will have the most recent time stamp, so it can be placed at the head of the linked list. When we need to expunge the oldest record, we simply go to the tail of the list (via a pointer) and remove that record. In short, this structure permits fast access of stored data and expunging on an LRU basis.

For optimal performance, it is not pointer caches or files themselves that are moved around when using chained hash tables. Instead, pointers are kept to files on disk, so deleting and adding entries is a quick memory-only operation until the file itself needs to be expunged.

APPENDIX B: TREE SEARCH ALGORITHM FOR USERS

The algorithm below allows a user to find a functioning primary server deterministically given a file name (K_p). It assumes that primary servers may not be able to respond for various reasons, either outage or busyness. It also assumes a top branching factor of 5 and a general branching factor of 3. The constant search_factor determines how deep a user is willing to search.

```
Find_primary (name){
   level = 5;
   n = number of a random node at level 5 (200..605)
   While (level > 0) {
      e = Lookup(name, n)
      Ask e if it can provide a pointer.  If it can {
         Return the pointer info and e
      }
      n = floor ((n-3) / 3) // get parent of node
      level = level - 1
   }
   Return “No primary server found”;
}
Lookup (name, n){
   Return Chord(SHA[name + n]) // + is concatenation
}
```

A user starts at the bottom of a document’s tree (level 5) at a random node, checking if the node has a pointer. If it doesn’t, the user travels up to the node’s parent and repeats the process. If the top-level nodes don’t have the file, the user can choose to start the search over again. If the randomization algorithm for choosing initial n is somewhat changed, we can guarantee that the user will walk up the tree and query each level 1 node (likely to be a primary storage server) at least once.

APPENDIX C: SUMMARY OF WAT COMMANDS

Author to Server

Name	Arguments	Description	Server Action
Insertable?	File name, n, size	Asks server if it can take another document as primary storage.	Checks if enough space is available.
Insert	Name, n, data, time stamp, version	Places file in server as primary storage. Only called if Insertable? returns true	Stores file in primary storage. Creates pointer cache for file.
Update	Name, n, data, time stamp, version	Update document in server	Updates document, purges pointer cache, asks children to update.

User to Primary Server

Name	Arguments	Description	Server Action
Get-SS	Name, n	Asks for a list of secondary servers that have document. Used in upwards tree search algorithm.	Replies if possible with a list of 10 SSs. Might include self initially. Drops request if busy.
Report-SS	Name, n, IP, "Off"	Reports to PS that SS is not functioning.	Deletes SS with given IP from pointer cache after checking with SS.

User to Secondary Server

Name	Arguments	Description	Server Action
Get	Name	Asks for a file	Sends if possible, otherwise indicates busy or not available. User reports to PS that it is now a secondary server.

Secondary Server to Primary Server

Name	Arguments	Description	Server Action
Report-SS	Name, n, IP, "On"	Report to PS that self has a given file. Done on activation or recovery.	Adds SS to pointer cache.

Primary Server to Primary Server

Name	Arguments	Description	Server Action
Replicate	Name, n, [pointer data]	Parent asks child to replicate pointer cache when parent is either full or notices high demand.	Child accepts if not not already serving as primary server. Child sends pointing-to messages to SSs.
Request-replication	Name, n	Child asks parent to replicate to it.	Server sends replicate message.
Update	Name, n, time stamp, version	Parent tells child that document has updated.	Purges pointer cache, asks children to update.
Purging	Name, n, [pointer data], "0"	Child tells parent that it is expunging a given pointer cache.	Parent adds child's pointer cache to own. Tells its own parents of purge. Parent sends pointing-to message to SSs.
Purging	Name, n, [pointer data], "1"	Parent of a purging child telling its parents about purge.	Parents add pointer cache to own, tells parents. Does not send pointing-to messages.

Primary Server to Secondary Server

Name	Arguments	Description	Server Action
Pointing-to	Name, n	Tells SS that it is now acting as the PS	SS updates PS pointer for file.

Appendix D: Table of Contents