Spankster: a Distributed Peer to Peer File Sharing System

6.033 Design Project 2

Kailas Narendran, Andrew Lamb, Alex MeVay
Saltzer/Clarke TR2
05/10/2001

Abstract

Spankster is a peer to peer file sharing protocol that is completely decentralized, and will scale to millions of users. A deterministic chunking algorithm allows clients to locate the disparate pieces of a file without relying on control messages. Controlled replication of files ensures file persistence and availability, even when nodes fail and at times of large demand. Spankster uses public key cryptography to ensure file integrity and authenticity, and to prevent unauthorized file changes. Spankster builds on the Chord peer to peer lookup protocol[1] to provide these additional features.

---

1. Stoica, Ion; Morris, Robert; Karger, David; Kaashoek, M. Frans; Balakrishnan, Hari. *Chord: A Scalable Peer to Peer Lookup Service for Internet Applications*. http://pdos.lcs.mit.edu/~kaashoek/chord.ps

Introduction

Overview

Spankster is a protocol for a peer to peer distributed file system. It allows any user to publish a file, and any other user to retrieve the file without relying on any central authority. Files only persist for a finite amount of time before they are purged from the system, unless the publisher notifies the system regularly that the file should be maintained. Clients may enter and leave the system at their convenience and can control the amount of their local network and storage resources that they contribute to the system. Publishers are uniquely identified with a public/private key pair. All files are signed with the private key and named using the public key of the publisher to authenticate their source. Only someone with the private key corresponding to the public key on a file can insert that file and renew its time to live.

Goals

- The primary goal of the Spankster system is to provide a file sharing service, similar to Napster[1], except that there are no central servers, and therefore no central points of control or failure.

- When a user retrieves a file from the Spankster system, the user should be sure that the file returned is the same file that was inserted by the publisher.

- When the demand for a file grows, the system should be able to dynamically adapt to deliver the maximum possible performance.

- Spankster tries to make a reliable storage system out of unreliable and untrusted nodes. Spankster should be able to continue to serve files indefinitely if nodes fail sequentially with a reasonable amount of time separating failures. If 5% of the Spankster nodes failed simultaneously (a power failure hits California, for instance), there should be only a one in a million chance that any particular file is lost.

- The system should scale to millions of users.

Assumptions

The primary goal of Spankster is to allow peer-to-peer file sharing without any centralized control. In order to accomplish this goal, Spankster becomes vulnerable to some forms of attack which are easily avoided using a centralized system.

A malicious user might launch a denial of service (DOS) attack by flooding the Spankster network with useless files. Preventing malicious flooding requires either restrictive per-user limits or an authority to decide which files are “good” and which files are “bad.” Both of these preventative strategies go against the goal of providing a user-driven, decentralized system. Another strategy to keep trash out of the system is to give individual users the ability to not participate in storing files

1. http://www.napster.com

they deem “trash.” However, because of Spankster’s reliability, most users would have to cooperate to remove any particular file from Spankster, and that cooperation would need to be coordinated from central location.

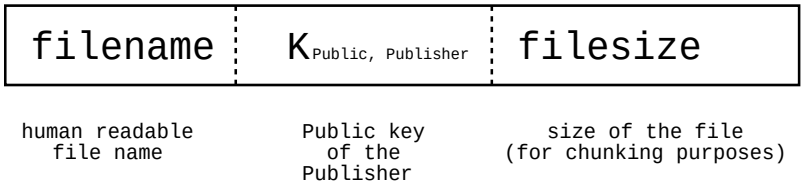
Spankster is primarily a protocol, not a program, and DOS attacks can also be initiated at lower levels within the protocol stack (such as TCP/IP). It is not feasible or advisable to try to prevent low level DOS attacks within Spankster or with most other end-to-end protocols.

Thus, the key assumption of Spankster is that the users cooperate. Cooperation is implicit even in the base Chord protocol, since the probability any one user stores his own file is virtually zero. In any system based on Chord, a user does not help the availability or persistence of his own files by joining the network, so any offer of services to the Spankster network is a purely philanthropic act.

Security and Naming of Documents

Spankster uses file naming for several tasks including authentication and garbage collection. In addition to the descriptive name of the file (dp2.txt, for example), a Spankster filename contains the public key of the author and the size of the file. The Spankster client uses the file size to locate particular chunks of a file that Spankster may have fragmented during an insert operation (Spankster locates files using a deterministic algorithm, described later). Spankster authenticates files using public-key cryptography. Every chunk of every file is signed with the author’s private key. Thus, a client that receives a file may check that the file indeed matches the name he was searching for by checking that the file contents verify with the key in the filename. With this system working correctly, the user must only trust that his rendezvous with the author to receive the filename was authentic.

Figure 1: Spankster filenames



Spankster filenames are long, ugly creatures, and are best hidden in a layer invisible to the user. The envisioned Spankster interface is web-like, and Spankster names are embedded in web-like links. We assume that users will hear about files much the same way most people hear about webpages, by someone else sending them the link, or through some type of search engine. The parts of a Spankster filename are shown in Figure 1.

Deterministic File Chunking and Replication

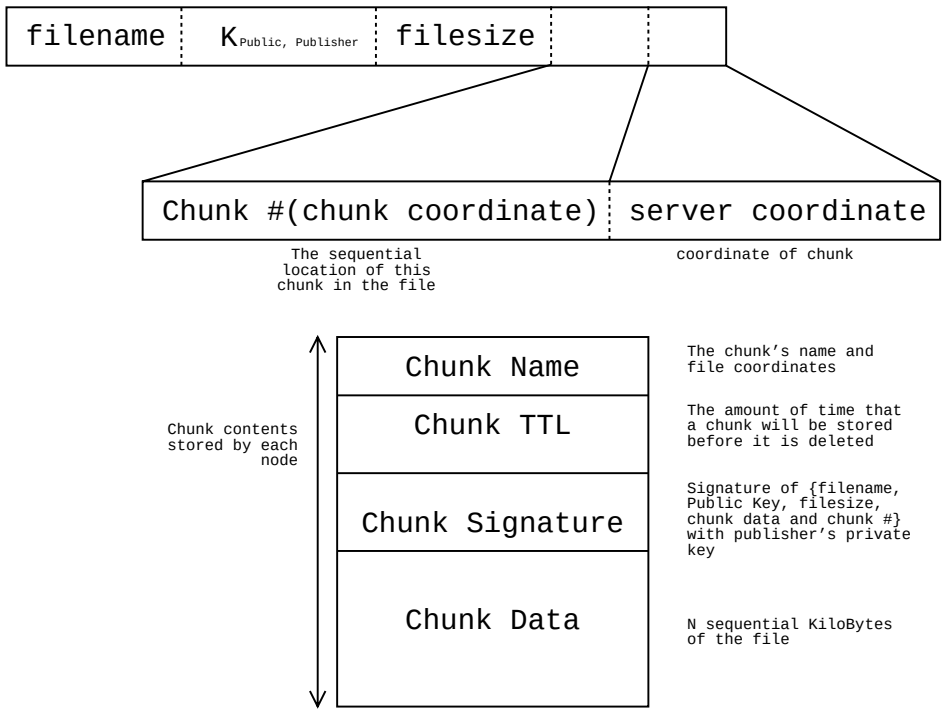
Purpose

Breaking a file into multiple chunks and then distributing each chunk among multiple hosts allows Spankster to provide high availability and high performance. Splitting files into chunks (chunking), by itself will increase availability while also increasing the likelihood of file loss. To provide high reliability, Spankster replicates each of chunk among multiple hosts and is thus able to achieve high fault tolerance. To increase performance, clients can request multiple chunks of a file

at once. The client could therefore (in theory) receive the file's constituent data chunks at the same time, at his own available bandwidth, even if the chunks are stored on computer with slower connections.

Overview

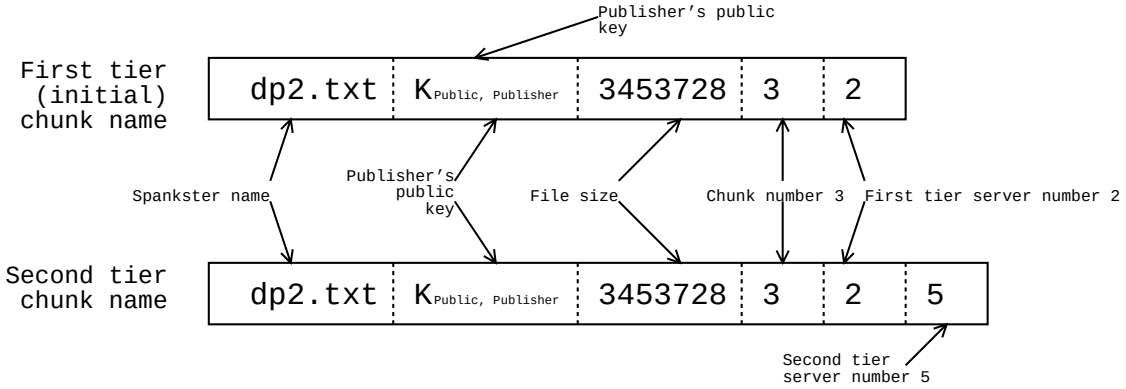
Figure 2: Spankster chunk names and contents



Each file inserted into the Spankster System is broken up into small, N KB chunks. R replicas of the chunks are created. Spankster names each chunk according to an established, deterministic scheme based on the main Spankster filename. The computed chunk name is then hashed to determine an associated Chord ID. The publisher signs {filename, chunk number, chunk contents} with his private key, and the signature is included with each chunk. By simply knowing the name of a Spankster file, the names of the file's constituent chunks can be recreated, and thus the Chord IDs of the chunks can be determined.

Spankster names file chunks by appending "file coordinates" to the original Spankster name for the file. File coordinates consist of the sequential number of the chunk within the file, and the "server coodinates" of that chunk. Server coordinates are a string of numbers between 0 and (R-1), which are used to make chunk names unique, and thus stored on different servers according to the Chord protocol. The following sections describe the generation and use of file coordinates in more detail. The Chord ID for the chunk is computed by hashing the chunk's new name with SHA-1. Figure 2 shows how a chunk name is constructed, and shows a generic chunk name, Figure 3 shows an example chunk name.

Figure 3: Example chunk name



Choosing N and R for fault tolerance and download performance

The calculations below suggest that choosing the values N = 100 KB and R = 6 is reasonable for the Internet of today.

Both N, the size of eah file chunk, and R, the number of replicas of each chunk, critically affect the performance of the Spankster system. The optimum values for each parameter depend on Spankster's environment and design requirements.

Most files are unusable if a random 100 KB is missing from them. It is therefore assumed that a file in the Spankster system is totally lost whenever a single chunk is lost. The fewer chunks (larger N) a file is split into the lower the probability of losing a file. However, greater the number of chunks (smaller N) a file is split into, the more evenly the load for retrieving the file is spread over all of the Spankster nodes. By choosing a value for N, performance and reliability can be traded off agianst each other.

R is the number of times that a particular chunk is replicated. The greater R is, the less the effect of random lost nodes (because R nodes must fail simultaneously for a chunk to be lost) but the greater the storage resources that are demanded of clients (by a factor of R over the initial file). The Spankster goals provide a starting point for choosing N and R. Spankster's reliability goal is to be able to lose a random 5% of its nodes, and not lose any files, and a file is considered lost if any one of its chunks is lost.

If a fraction L of the nodes are lost, the probablility that any chunk is lost is L^R. The probability that all chunks survive is

$$(1 - L^R)^{\left(\frac{S}{N}\right)} = P(allChunksSurvive)$$

where S is the size of the file, so that S/N is the number of chunks a file is broken into. As expected, the smaller N is, the greater the probability of failure, and the greater R is, the lower the probability of failure. Since N directly affects the download speed of the system and disk space on personal computers is growing rapidly, N is chosen for good download performance, and the above equation constrains R.

100 KB was chosen for a value of N because it is an easily digestible size for most computers connected to the internet today. At present, we assume that the slowest possible computer in the Spankster system is a home user connected over a 56.6 Kbit/sec modem. At this rate, allowing for packed headers and noisy transmission lines and assuming full use of bandwidth, it would take around 20-30 seconds (14.13 seconds if at full speed; 30 seconds might be a more reasonable estimate) for any particular chunk to be transmitted. As broadband connections (eg cable modems and DSL lines) continue to proliferate, the time to transmit a 100 KB file will continue to decrease. Using a broadband connection, the transmittion time is on the order of 1 second.

Since a client can request a file's chunks simultaniously from multiple different Spankster hosts, a hypothetical client with unlimited bandwidth could download a file of any size in about 30 seconds or less. To determine R, we need an estimate for a typical filesize. If the popularity of Napster is any indication, much of the traffic on Spankster will consist of MP3 music files, which have a typical file size of about 5 MB. By setting N = 100, S=5,000,000 and and P(fileLost)=1-P(all chunks survive) = 1/1,000,000 = 0.000001 R is constrained to be at least 5.15, so R should be set to 6 to achieve Spankster's reliability goal. Even this is conservative, because Spankster clients will make intersession disk storage possible (described in the section about joining the network below), which will prevent most crashes from causing a node to permanently lose any data.

Protocol support of Replication and Chunking, and Load Sharing

File coordinates

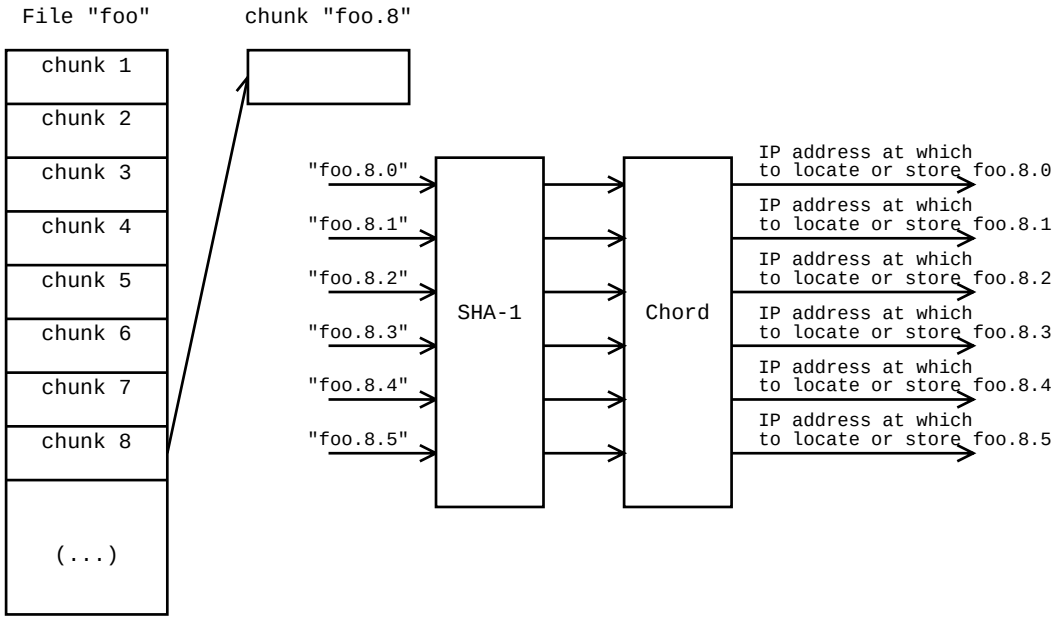
Spankster manages file replication and chunking by appending the file coordinates (the chunk number and server coordinates) of each chunk to the end of the original Spankster filename, and inserting the chunk into the system under the new name. This allows Spankster to locate the chunk replicas on different servers (since the hash to Chord IDs is consistent, and spreads the storage load evenly), at locations that can be determined with only the main Spankster name.
For the example shown in Figure 4, chunk 8 of file "foo" might be found at foo.8.1,or perhaps at foo.8.4.3.5. The first number after the filename indicates the chunk number, and the successive numbers indicate the host serving the file. When a file is inserted into Spankster, each chunk is inserted R times, where R is the replication number. Chunk foo.8, for example, would be inserted as foo.8.0, foo.8.1, ..., foo.8.(R-1) The first server coordinate is always present and accomplishes replication for fault tolerance. Additional coordinates add additional replication for load balancing, as the next section illustrates.

File coordinates and high network traffic

If any of the Spankster nodes serving files becomes overloaded with traffic, the overloaded node makes extra copies of that chunk available, and then implicitly redirects requests to the new copies. A server hosting foo.8.3, for example, it would reinsert the file another R times at new locations foo.8.3.0, foo.8.3.1, ..., foo.8.3.(R-1), and begin redirecting all traffic to this addtional "tier" of new copies. If any of the nodes at which the copies were inserted themselves become overloaded, the overloaded node would add an additional field to the name, and reinsert the copies in the same manner, with filenames such as foo.8.3.5.1. This process can continue indefinitely until all the network traffic is handled. These names logically suggest a tree, but that analogy is misleading. Since the leaf addresses are deterministic, it is not necessary to traverse any branch nodes

to get to a leaf. Indeed, when a node wishes to redirect a client node to another tier of copies, it has to reply only with a “No” message (the implicit redirect packet), and the client will automatically know where to look for the additonal copies of the chunk, simply by appending a new field to the previous file coordinate.

Figure 4: How publishers and clients locate chunks



File coordinates have two main advantages. First, implicit redirect packets are very short, helping an already overloaded node. Second, even if a leaf does not reply, the requesting client can still determine where to locate chunks at lower tier nodes.

Retrieving documents

Consider the following example of executing the command, `get(“foo”)`: To retrieve chunk foo.8, (one of the steps in the command) the originating computer picks a random number between 0 and R-1, say 4, and requests that copy, foo.8.4 from the IP address returned by handing Chord the hashed version of foo.8.4. If the initial request times out, the client picks a different number at random, and requests that chunk copy number(for example foo.8.2).

The client will continue requesting the same chunk with different coodinates until one of four things happens. First, the `get` might succeed, and the client retrieves the desired chunk. Second, the client receives an explicit redirect. Third, the client receives an implicit redirect packet (a “No”). Fourth, the client does not receive a reply from any of the possible hosts.

In the second case, the client searches at the address specified by the explicit redirect. This would be necessary if the original host had insufficient storage space for the chunk.

In the third case, the client chooses another number at random, appends it to the current server coordinate of the host who sent the “No” and starts looking for the chunk on the next “tier” in the same manner as the original search.[1]

The fourth outcome, no replies, is not as bad as it seems. It means either the file is not present, or all nodes hosting the first tier of the file are receiving too much traffic even to send enough implicit redirects. The client will assume the latter, and that the nodes have reinserted the file on lower tiers. The client will now by default search for the chunk at locations in the one tier down, foo.8.X1.X2, where X1 and X2 are random numbers between 0 and R-1. If the file is not found, this search will continue on lower tiers (to foo.6.X1.X2.X3.X4..., etc.) until the client finds the file or decides to give up, either by exceeding a maximum search depth, or a maximum search time specified by the user.

If the any file chunk is not found, the `get` command returns an error. Otherwise, all of the file data is extracted and reassembled from the chunks. During this process, the private key signature of each chunk is verified using the public key in the filename, thus ensuring that the file is authentic. If a chunk does not verify, the client requests a new copy of the chunk from a different location.

Traffic Loading Analysis

Replication to meet changing demand

The `get` protocol above functions efficiently even with low-bandwidth hosts. Since the traffic on each tier is only about 1/R times what the traffic would have been on the redirecting node, popular files may be replicated to reduce the data loading at all nodes to an acceptable level with only a small number (logarithmically growing) of redirects. The limiting factor in download performance is not data loading but sending implicit redirects.

Justification

Some estimates will support the claim that the get protocol works with slow hosts: We assume that since this is a file-sharing system, users will only download any file once. Also, we assume that a file will be popular over the course of a week (smaller times here are conservative in the following calculations), as news of the file spreads. Assume, conservatively, that an implicit redirect packet (the “No” message) is 1000 bits, including all headers and transmission overhead. In the worst case, all 6 nodes on the first tier are connected by 56.6kbit/sec modems for the whole duration of the analysis, and traffic is so high that they send only implicit redirects. We assume now that traffic from other files in negligible and that the request load is constant. Each second, R*56.6k/1000=339 file requests can be processed. Over the course of a week, this is 200 million file requests. Some of the assumptions above are not so conservative (constant load, each user downloads a file once), but even so, these calculations show that even in the worst case, the system is still capable of serving millions of users in a reasonable scenario.

---

1. There is another possible way to proceed after receiving an implicit redirect. Instead of searching the next tier of hosts (of the form foo.6.X.X), a client could keep trying the other nodes on the first tier (of the form foo.6.X). However, if requests load balance evenly across all nodes on a tier, as they should, nodes on a lower tier should be receiving only 1/R as much traffic as their parents, and hence are a better place to look for the desired file chunk.

Inserting files

Insertion

The publisher of a file is solely responsible for insertion and deletion of her files from Spankster. Chunking is done by the client software of the publisher according to the Spankster protocol, so all Spankster nodes store chunks rather than whole files. The publisher's Spankster client determines the IP address of the host that should store any particular chunk (that chunk's storage host) by performing a Chord lookup on the hashed chunk name as described above. The publisher's client then sends a timestamped "Insert" message, signed with the author's private key along with the chunk contents to the chunk storage host. The receiving host can use the signature on the "Insert" packet and the public key in the chunk name to verify that the author is submitting this file. If validation fails, an "Error" is returned to the publisher and the chunk is discarded.

Table 1: Packet types

Packet Type	Description	Data
AddChunk	Receiving node stores the chunk	Chunk Name, Chunk Data, Chunk Signature, Time To Live
GetChunk	Request for a particular chunk	Chunk Name, Return Address
Receive-Chunk	Response containing a requested chunk	Chunk Name, Chunk Data, Chunk Signature
No (implicit redirect)	Used to implicitly redirect traffic to next tier of nodes	(None)
Explicit Redirect	Used to explicitly refer a GetChunk request to another Spankster host.	Chunk Name, New Server Address
KeepAlive	Resets the TTL timer for the specified Chunk	Chunk Name, timestamp, signature

Time to live

Each chunk name contains a time to live (TTL). When a host receives a file, it is time-stamped locally and should be deleted after the specified time. It is the responsibility of the publisher to issue a "KeepAlive" message to all nodes containing chunks of his file. If the individual that inserted a particular file deems it unnecessary or obsolete, there is no need to keep the file around, so chunks with expired TTLs are deleted.

To ensure that infinite TTLs are not set by lazy authors, the maximum time to live allowed by Spankster is four months. The average software programmer writes about 10^1 bug free lines of code per day. A "large program" is defined as having more than 50,000 lines of code. We will assume that most software on our network is on average about 25,000 lines of code and the average size of a medium size team is 10 programmers[2]. If they were to rewrite half of the software

1. http://www.vni.com/books/press/sales_CNL00.html

between releases, the time period between releases would be about 125 days. This time period is a very rough prediction of system usage.

When chunk of a popular file is copied to the next “tier” of Spankster nodes, the maximum TTL at the next “tier” is only 48 hours. After 48 hours, if the redirecting node is still hosed it must send KeepAlive messages to the next “tier.” The time limit of 48 hours gives users around the world sufficient time to get a file the day it is released.

Because each chunk has a time to live which it is the publisher’s responsibility to update, Spankster does not provide an explicit delete operation. To delete a file, the publisher merely lets the TTL expire.

File updates

To preserve file consistency, Spankster does not allow file updates. Keeping all of a file’s chunks consistent throughout an update operation is very hard. The publisher would have to invalidate all chunks, ensure that they all were invalidated, and then update them. With unreliable nodes that can effectively enter and leave the system randomly, it is not clear that this process for updating file contents would work in all cases. Since Spankster users get filenames from a rendezvous with the publisher anyways(eg a web page), to update a file’s contents the publisher simply inserts the file into Spankster under a different name and changes the pointer he gives out to people looking for his file.

Entering and Leaving the Network

A node joins the system

When a node joins the network, it must alert other nodes of its presence and take over its share of files. Spankster calls the `join()` function of the Chord system, which updates and propagates the necessary Chord address information througout the Chord system. The new Spankster node then contacts its own Chord successor, and by comparing Chord IDs, determines which chunks it should take over from its successor.

The next step depends on the user settings of the Spankster program. In Spankster, the user is able to determine how much disk space to allow Spankster to use, and also whether that disk space is made available again after the program closes. If the user has chosen to allow Spankster to store files on the disk between sessions, there is a good probability that many of the files that the new node has been alotted from its successor are already on its disk. This locality occurs because chunk and host IDs do not change, and thus chunk to host assignments do not change often.

The recently joined node compares its files cached on disk with those it has been assigned to host, and receives any new files. This disk caching significantly saves transfer overhead when nodes join and leave the system. If the user has not allowed Spankster any intersession disk space, all of the relevant chunks must be transferred to the new node from its successor. The disk cache is managed by an ordinary chaching policy, such as LRU (replacement of the Least Recently Used file).

---

2. http://www.hpl.hp.com/techreports/91/HPL-91-170.html

Periodically, and before any chunks are removed from the cache, a Spankster node will perform a "GetChunk" on the first "tier" of nodes for each chunk about to be purged, and each chunk it is hosting. If any of the "GetChunk" requests return an error, the node will assume that some failure has occurred and the chunk at that server coordinate has been lost. The node will reinsert the chunk at the appropriate coordinate using its own copy. This self-healing process will ensure that Spankster does not lose files over time.

Joins and inserts with limited resources

When a node joins the network, it might not have enough space to store the chunks that Spankster allocates to it. In this case, for each of the chunks it cannot store, it saves a pointer to the current server of the chunk, and the file remains with the successor. If a node does not have enough space to store the requisite pointers, it is not allowed to join the network.

If a node runs out of space, but is asked to store a chunk, it remains responsible for that chunk. The node will proceed in a manner similar to what it would do if one of its chunks were receiving excessive traffic. The node will insert the chunk, say foo.8.3, at the first lower-tier node that has space to store it (foo.8.3.0, or foo.8.4.5.2, for example). When it receives requests for that chunk it responds with an explicit redirect message which gives clients the correct address for the requested chunk.

Leaving the network

Spankster's shutdown/leave mechanism is very similar to its join procedure. When Spankster is shutdown properly (i.e., not by a system crash or network disconnect), the Spankster node passes on all the chunks it had been storing to its successor, (except for any that the successor might have cached on disk). The node then calls the Chord `leave()` command, which makes the appropriate updates to the address lists within the Chord network. The chunks left on disk either remain for the next session, or are deleted to free up disk space, according to user preference.

Conclusion

Spankster sets for itself the ambitious goal of providing distributed file sharing with absolutely no central control. To accomplish this goal, we make the dubious assumption that malicious denial of service attacks will not occur. Any sort of trash supression scheme must involve some sort of central control, which explains the paucity of truly peer-to-peer networking protocols. Spankster uses many novel approaches such as deterministic chunking and distributed hashing. As these technologies are further refined in the coming years, hopefully systems like Spankster with no central control will continue to proliferate.