

Runtime Quality Attributes of Distributed Storage Systems

by Zef Hemel

Abstract: In this essay, runtime quality attributes of distributed storage systems are discussed. These quality attributes are scalability, security, robustness and performance. For three different distributed storage systems (GoogleFS, DISP and Gnutella), the architecture and fulfilment of these quality attributes are looked at and evaluated.

1. Introduction

The need and use of large-scale distributed storage has rapidly increased in the last few years. Well-known examples of this are the Google storage system that stores multiple copies of the whole Internet, and peer-to-peer networks like Gnutella, Napster and Kazaa through which terabytes of content are shared.

Because of the rapid growth of these systems, it becomes interesting to look at how different kinds of distributed storage systems deal with certain architectural quality attributes like scalability, security, robustness and performance. The purpose of this essay is to show how different distributed storage techniques (designed for different purposes) work (on a high level) and fulfil these quality attributes. The storage systems that this essay will attempt to shed some light over are:

- DISP, the Distributed Information Storage Protocol, a research project that proposes a pure client-server implementation for distributed storage.
- GoogleFS, the storage system that was created by Google to store their massive search index and is starting to be used for other kinds of storage, such as mail, as well.
- Gnutella, one of the first fully decentralized peer-to-peer distributed storage systems (used mainly for file exchange).

2. Quality Attributes

Before we look at the different distributed storage techniques, we first define the quality attributes that we'll be looking at.

2.1. Scalability

Scalability indicates the capability of a system to increase performance under an increased load when resources (typically hardware) are added. A system whose performance improves after adding hardware, proportionally to the capacity added, is said to be a scalable system.

There are two kinds of scaling:

Vertical or scaling means to add resources to a single node in a system, such as adding memory or a bigger hard drive to a computer. To scale horizontally or scale out means to add more nodes to a system, such as adding a new computer to a clustered software application. [1]

In this essay we'll talk mainly about horizontal scaling.

2.2. Security

Security, in this context, deals with enforcing that users or programs can only perform actions that they are allowed to perform, so that, for example, nobody can read or manipulate data that he or she should not have access to.

2.3. Robustness

Robustness is the resilience of the system, especially when under stress or when confronted with invalid input. In the context of storage systems it in particular deals with fault-tolerance. If a node in the storage system breaks down, no data should be lost and the system should still continue to work. Also, if a data file is corrupted, this should be fixed, or at least detected by the storage system.

2.4. Performance

Performance deals with the speed of response and the throughput (from the location where the data is stored to the requester) of a storage system.

3. DISP

DISP, of the three discussed storage systems, is probably the least well known. To be quite honest, it is even unclear if it's actually being used in real-life applications today. However, it was design specifically to satisfy a couple of quality attributes that we're interested in. These are: efficiency (as in performance), security and fault-tolerance (robustness).

DISP was developed by Daniel Ellard of Harvard University and James Megquier of Gnuterra Corporation.

DISP uses the client-server architectural style. Most other distributed storage systems also

require some server-server and client-client communication, but DISP doesn't. This makes DISP more scalable, its creators say.

3.1 Assumptions and Restrictions

DISP assumes that the network in which data is stored (the servers) is relatively static. It performs best if the network remains unchanged during operation and servers work correctly. It is able to handle changes, if needed, however.

Objects in DISP are immutable, which means that they cannot be changed once stored in the system. It is, however, possible to create new versions of an object. All versions of an object remain accessible.

DISP assumes there's a authentication system already in place that is able to authenticate users and sign messages.

3.2 Writing Data

Storing data is done using the Information Dispersal Algorithm (IDA). This algorithm is a more general form of RAID-5, where one data object is split up into multiple redundant shares and stored on multiple servers. This ensures that once a server or hard disk fails there's still another copy available somewhere else on the system.

Storing data on the system is a three-step process executed by the client:

1. Obtain a WriteHandle. A client needs a certificate (WriteHandle) to prove that it is allowed to store a particular object on the system. After an authentication procedure, it obtains this WriteHandle from a server.
2. Prepare the redundant shares of the to-be-stored object and send it to the servers.
3. Commit the object. Once storage confirmation has been received from all the servers a commit message is sent to all servers.

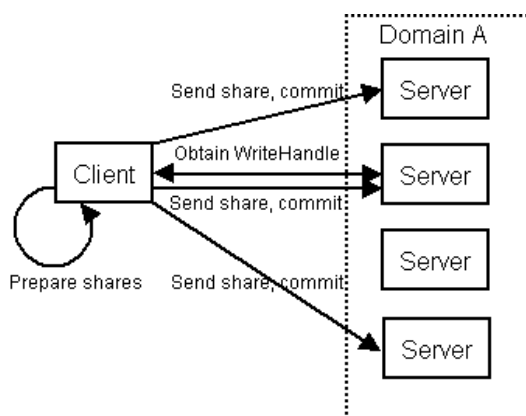


Figure 1: Writing data in DISP

3.3 Reading Data

When a certain object has to be read, a two-step process starts (from the client):

1. Request ReadHandles from servers. The client requests a list of all shares of a particular object that the server has.
2. Gather the shares. Once the client has found all the shares of the object from different servers that it needs it will obtain the shares from these servers.

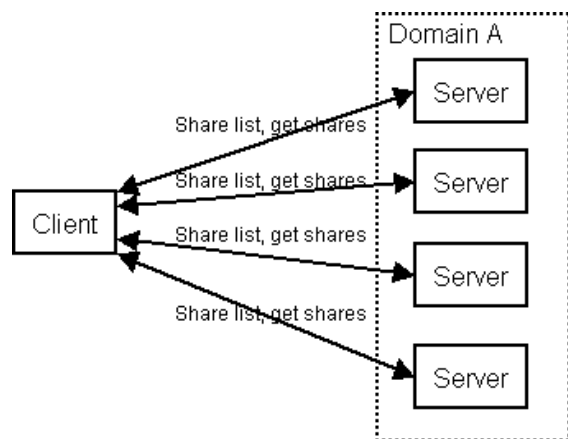


Figure 2: Reading data in DISP

3.4 Quality Attributes

DISP satisfies the discussed quality attributes as follows:

Scalability: servers do not talk to one another; they only handle requests from clients. Therefore it doesn't matter if there are two or fifty thousand servers in a network, the servers will work just as fine. Problems may occur for the client, however, if server domains become too big. When storing and retrieving objects, clients have to guess where an object can be stored or retrieved from. According to the DISP specification it will do this by contacting servers within the domain, for as long as all the shares of an object are gathered. This means that a client at least has to know in which domain an object is stored. If a domain is very big (many servers), finding all the shares of a file can take a long time. It is therefore important that domains remain relatively small.

Security: DISP allows for shares to be stored in an encrypted manner. The keys are stored in a similar distributed manner as the data. Authentication is not part of DISP, however is assumed to be in place. All messages sent between client and server are encrypted.

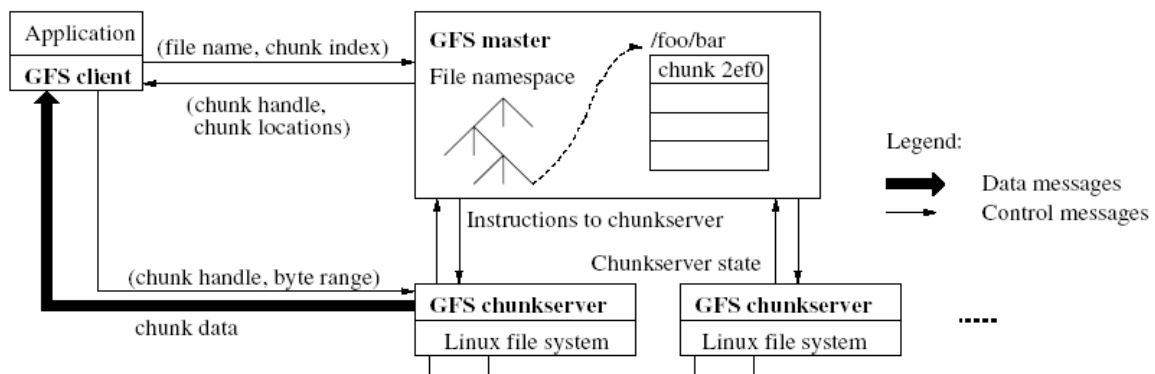


Figure 3: The GoogleFS architecture [3]

Robustness: DISP redundantly stores each piece of data on the system. This means that multiple copies of all data are stored on different servers. If one server fails, or the data is corrupted this can be discovered and fixed. When data is stored on the system, a check vector is stored with each share. This check vector holds checksums for all shares. How many times a particular share is stored is up to the DISP client, it is something that can be configured.

Performance: DISP is a system that allows for many different set-ups and configurations. Encryption algorithms used, using check vectors or not and using secure communication or not greatly impact the system's performance. Depending on the needs within an organization it is best to evaluate which configuration should be chosen and what its trade-offs are. The DISP paper [2] gives performance results for different configurations.

4. GoogleFS

Google is the most popular search engine today. It has the biggest search index and handles enormous amounts of search requests at an incredible speed. Since the Google IPO, lots of new Google projects started. Google started to offer free 1GB mailboxes, stores and searches video and much more.

The question, of course, is how Google manages to handle all this storage. It is clear that there's a massive storage system necessary in order to manage all the data that Google has to keep. Already quite early in the process Google engineers realized that they had to invent a cheap, yet efficient solution to this problem. For this they developed GoogleFS (the Google File System). [3]

4.1 Assumptions and Restrictions

Google runs thousands of cheap commodity hardware servers to store their data on. Many break after a while, but, as with DISP, all data is stored on multiple servers. GoogleFS was specifically designed to handle corrupted data and broken servers, because it's bound to happen; or as they phrase it themselves "component failures are the norm rather than the exception".

Second, the GoogleFS is designed to deal with very big files, usually several gigabytes big. It was optimized specifically for big file sizes like these.

Third, appending to the end of a file is the most used operation (beside reading). Current data is not usually changed a lot and data is not removed much either. Therefore the GoogleFS has been specifically optimized for the reading and appending operation.

4.2 Architecture

The GoogleFS consists of one master server, multiple chunk servers and clients (see figure 3).

The master server stores:

- The file namespace (file system structure)
- The file-to-chunk mappings
- The locations of chunks

Data files are split into chunks (of 64MB each). Each chunk has a 64-bit identifier and is stored at at least three servers, preferably located as far from each other as possible (topologically).

All this data is stored in-memory data structures for maximum performance. The file namespace and file to chunk mappings are also stored on disk, and replicated to back-up servers. The locations of all the chunks

however are obtained from all the chunk servers on master server start-up.

The master server also tracks which servers are still up and running by periodically sending them heartbeat messages (which the chunk servers should respond to).

Chunk servers store data chunks and serve them to clients asking for them.

Clients are machines (which simultaneously can be chunk servers) that request data and want to store it (for example indexers or people doing Google search engine searches).

4.3 Writing Data

For a client to be able to append data to a file, it first has to ask the master server for the chunk server that has a lease on that data and the locations of all the replicas. The chunk server that has a lease on a chunk manages that particular chunk and all manipulations of it; it makes sure that all manipulations are applied on all replicas in the same order. The reason that the master server doesn't do this is to keep the load of the master server.

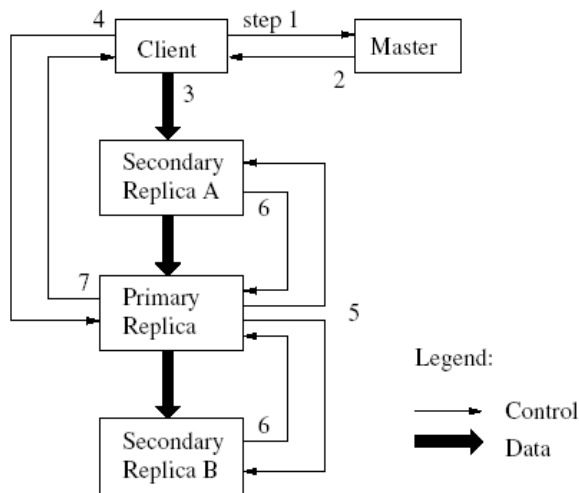


Figure 4: Writing data in the GoogleFS [3] After the client knows about the replicas and primary replica (the chunk server that owns the lease), it will start pushing the data to the nearest chunk replica (no matter if that's the primary or just a random replica). In order to use network bandwidth as efficiently as possible, the receiving server will then on its turn push to the replica most near to it and so forth. Once all storing chunk servers have received the data, the client sends a write message to the primary replica. The primary will forward this message to the other replicas and wait for confirmations, if all are confirmed; the client receives the final confirmation of the data being successfully stored.

To improve performance even more the data and operation flows are decoupled, which means that a client can already start pushing data to a replica before it's ready to store it. All this data will be kept in a buffer until it is needed.

4.4 Reading Data

To avoid the master server to become the bottleneck of the system, the master server is avoided as much as possible for reading data. When the client wants to read data it will ask the master server for chunk servers where the file is available and it will cache this information for a while. The retrieval of the data is done directly from the chunk servers and doesn't involve the master server at all.

4.5 Quality Attributes

GoogleFS satisfies the discussed quality attributes as follows:

Scalability: chunk servers, which store all the data, can be added and removed at any time. The master server will find them and use them to store replicas of chunks. Growing the amount of chunk servers can go on until the master server becomes the bottleneck. The master server is just one machine that manages all metadata and chunk locations. Much effort, however, has been put into keeping as much load off this master server as possible. When it can be avoided, it is being avoided. Also, the amount of metadata needed to be stored in-memory per file chunk is very minimal and the Google engineers don't expect this to be a problem in the near future. Google currently runs a GoogleFS system of thousands and thousands of servers and it seems to be scaling up very well.

Security: GoogleFS was designed to be run in a relatively closed environment and no real security measures have been taken in the sense of hacking. Data integrity, however, is being guarded very strictly with checksums and the like. The reason for this, however, is fault-tolerance (robustness), not security.

Robustness: the master server tracks which chunk servers store which chunks. Every chunk is stored at least three times in the system (although replication levels can be set on a per namespace basis). As soon as the master server finds out that a chunk server broke down, it will order to let the chunks on the server being copied to other servers to make sure enough copies of each chunk are secured. Chunks contain a checksum for each 64KB, so that it is easy to check if data has been corrupted. Chunk servers also do this

themselves while in idle periods. When a corrupted replica is detected it is removed and a clean one is requested from another chunk server.

Performance: GoogleFS does many things to get an as high performance as possible. One thing is the replication of data. Chunks are stored on multiple places at once and chunks that are read most get a higher priority for being replicated even more often. Checksumming is highly optimized for appending data. Also with the use of IP addresses (which in the Google data centres are related to location) data traffic flow is regulated. A write initiated by a client for example doesn't send its data to all chunk servers itself, but lets each chunk server forward it to its nearest neighbour which is more efficient in bandwidth usage. That the approach taken by Google is successful is known to everybody who once did a Google search. Even with terabytes, maybe even petabytes of data and thousands of people using it simultaneously, Google responds promptly.

5. Gnutella

Gnutella was first developed by Justin Frankel and Tom Pepper of Nullsoft, in early 2000, soon after the company's acquisition by AOL. On March 14, the program was made available for download on Nullsoft's servers. The event was prematurely announced on the web, and thousands downloaded the program that day. The source code was to be released later, supposedly under the GNU General Public License (GPL).

The next day, AOL stopped the availability of the program over legal concerns and restrained Nullsoft from doing any further work on the project. This did not stop Gnutella; after a few days, the protocol had been reverse engineered, and compatible open-source clones began to appear. This parallel development of different clients by different groups remains the modus operandi of Gnutella development today. [4]

Gnutella is an application that allows you to share your files and to search for files and download and search for file offered by other people. This type of peer to peer file sharing, which is similar to what applications like Kazaa, Napster and eMule do, is most popular for people illegally downloading music, but has legitimate uses as well.

Unlike DISP and GoogleFS, Gnutella is a pull-only system. A client doesn't store data in the system, a node just stores its own data that other interested nodes can download from it.

The Gnutella was one of the first purely peer-to-peer file sharing protocols, which exposed some major problems these kinds of systems have. The nice thing is that it's an extremely simple protocol [5], which is easy to implement.

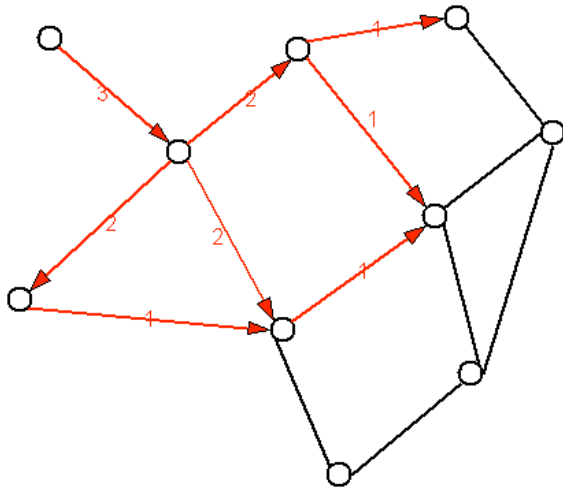
5.1 How it works

In order to connect to a Gnutella network you need the IP of one or more nodes that are currently already connected. Many of the Gnutella servers (which is the Gnutella term for the application which is both a client and server simultaneously) come with a list of IPs of nodes that are usually online. Once you're connected to a node you start receiving messages that you're supposed to respond to and pass on to your neighbouring nodes. You can also send messages yourself. In Gnutella there are five kinds of messages:

- **Ping:** you send this message to discover nodes in the network. Every node in the network that receives such a ping message is supposed to respond to it using a pong message. These pong messages are routed back to the node sending the original ping message but can be read by nodes that it is passed by. By intercepting pong messages you can add more node IPs to your list of IPs to connect to. To make the network stronger you are encouraged to connect to more than one node.
- **Pong:** this is the message that you reply to a ping message with, to make yourself known.
- **Query:** if you want to query for something, for example search for a file, you send a query message. Each node is supposed to respond to this message with a QueryHit message (if it has matching files) and pass on the query message to its neighbour nodes.
- **QueryHit:** the message to send back to the sender of a query when you got matching files.
- **Push:** this message initiates a connection with a firewalled node to make file downloads from firewalled nodes possible.

To stop messages from being passed through the network forever they contain a TTL (Time To Live) tag. On every node the message passes (called a "hop") this number is decreased by one. When the number is zero the message is discarded. For example, let's assume you choose 3 as the TTL for your message. The message flow would look something like this (the initial message is sent by the node at the top left, red arrows show the

message flow, lines mean connections between nodes):



With a TTL of 3 some nodes aren't reached. How many nodes are reached greatly depends on how the network is connected.

5.2 Quality Attributes

Scalability: this is Gnutella's main problem: it doesn't scale very well. The problem is that every message has to (ideally) be sent to every single node. Therefore the network is quite bandwidth-intensive. It works fine with a couple of dozen of nodes, or even a couple of hundred, but after that it's just unusable.

Security: this was not a design goal of Gnutella. All data is sent in clear text format and not much checking of data is done at all.

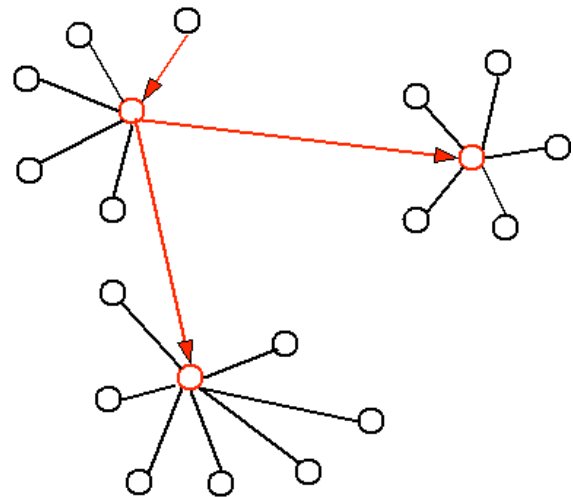
Robustness: the only thing that Gnutella does related to robustness is storing multiple copies of a file - at least, if more than one person has a particular file, which usually is the case, though never guaranteed. All file transfers have to be initiated by a user, so it totally depends on the popularity of a file whether it remains in the network or not. Gnutella doesn't do checksumming to check for file corruption.

Performance: it takes a lot of time for your message to pass through the network. When you search for some file it can take ages before you get useful response. As said before, Gnutella works fine for a small amount of nodes, but at a certain point it just becomes unusable.

5.3 Improving Gnutella

As soon as it became apparent that the Gnutella approach of organizing your network wasn't very scalable, other approaches were invented. Kazaa, for example, used the

concept of super nodes, which are nodes running the same software as any other, but which have more resources at their disposal (mainly bandwidth). Super nodes store an index of all the files that the nodes have that are connected to it, nodes also preferably connect to a super node rather than other normal nodes. Using this the network topography becomes much more efficient and scalable. For example a search can take place as follows (the red circles are super nodes):



6. Comparing

6.1 Usage

DISP and GoogleFS are two approaches to distributed storage that are fairly similar. They're both meant to be used by some application that wants to read and manipulate a huge pool data, similarly to a normal file system. Although GoogleFS focuses on reading and appending, DISP doesn't manipulate existing data objects at all, it only adds new ones and retrieves current ones.

Gnutella users on the other hand are interested in something totally different, they don't want to manipulate data, they just want to download it from others.

6.2 Scalability

Scalability in DISP is achieved through not having a master server. There can be any number of storage servers that a client talks to. As long as server domains remain small and the client roughly knows where to look for data objects, DISP can scale very well.

Even though the GoogleFS does use a master server, it only handles administrative tasks and it's used as little as possible. Thousands of chunk servers handle the actual storage. In

practice the GoogleFS approach seems very scalable.

Gnutella doesn't use a master server at all. However, because of that its scalability is really bad. For optimal effect of a message it has to be passed through the whole network (all nodes). This works fine if there are just a couple of these, but becomes very inefficient once the number of nodes grow. Adaptations of the Gnutella protocol propose local super nodes (local master servers) that index all files available in a cluster of nodes so that the message doesn't have to be passed to all individual nodes.

6.3 Security

One of the design goals of DISP was security. It assumes there's already an authentication system in place which it can use. Data can be stored in an encrypted format and communication can be done in an encrypted fashion as well.

GoogleFS is used in a protected and relatively closed environment and therefore doesn't take any security precautions.

Gnutella does not offer any security of any kind either. The protocol is unencrypted and it's very easy to see what other users are doing on the network (most Gnutella clients even show what people are searching for). For many people this is perfectly fine as no (to them) personal data is sent over the network.

6.4 Robustness

In the DISP system the client stores multiple copies of an object share on multiple servers. If one server dies the shares can still be retrieved from the other server. Checksums can be used to check if files are corrupted. DISP, however, is optimized for stable hardware that doesn't fail, unlike GoogleFS.

The GoogleFS system also stores multiple copies of data chunks on its system and checks for data corruption using checksums. On top of this, the system itself can detect when a server dies and automatically replicates the chunks that were on there to other servers so that the minimum amount of copies of a chunk is always available. Chunk copies are also stored as far from each other as possible, so that in case of, say, a rack of server catching fire the other copies are kept far from it.

The Gnutella system itself doesn't manage replication of data at all, nor does it do checksumming. Replication is done by its users that download the files they care about. This way the most popular files are replicated

the most and the files that nobody cares about can easily vanish from the network once the last holder of a file disconnects from the network.

6.5 Performance

The DISP system is highly configurable. You can switch on and off a lot of features like encryption and checksumming, which greatly impact DISP's performance. I'm not aware of any systems where DISP is used in practice, so how it performs in a real-life system remains to be seen. The benchmarks done by their creators seem promising, however.

GoogleFS, on the other hand, is an example of a distributed storage system that has proven itself in practice. Google stores multiple copies of about the whole internet and has hundreds, if not thousands of people searching through it simultaneously, still results are delivered within the second.

Gnutella's performance with any network bigger than just a couple of nodes is really bad. Everybody agrees on that and that's why people have worked on improved versions of the protocol, so far the results are looking promising. There's a Gnutella 2 protocol in the works and products like Kazaa show that it can be done.

7. Conclusions

The field of distributed storage systems is an interesting and diverse one. Because distributed storage comes with so many challenges (many of which not even mentioned in this essay), most distributed storage systems are developed for one particular kind of use. The DISP system for example doesn't really support manipulation of current data, just adding new data. The GoogleFS system is optimized for appending data to current data files. Gnutella on the other hand is a pull-only storage system. People that join the network have files stored locally and others can download it.

Because the storage systems are so diverse their fulfilment, or lack thereof, of the quality attributes we looked at is diverse as well. In systems like Gnutella and GoogleFS security is not dealt with at all; in DISP it is. Robustness is an example that DISP and GoogleFS deal with in roughly the same manner, but Gnutella doesn't deal with at all.

8. References

1. Wikipedia's scalability definition: <http://en.wikipedia.org/wiki/Scalability>

2. D. Ellard, J. Megquier, "DISP: Practical, Efficient, Secure and Fault Tolerant Data Storage for Distributed Systems", Harvard Computer Science Technical Report TR-17-03, December 2003.
3. S. Ghemawat, H. Gobiuff, S. Leung, "The Google File System", Proceedings of the Nineteenth ACM Symposium on Operating Systems, October 2003
4. Wikipedia's history of Gnutella section:
<http://en.wikipedia.org/wiki/Gnutella>
5. Gnutella protocol specification version 0.4:
http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf