Chapter 4

Peer-to-Peer Storage

4.1. Introduction

Peer-to-peer storage applications are the main actual implementations of large-scale distributed software. A peer-to-peer storage application offers five main operations: a lookup operation to find a file, a read operation to read a file, a write operation to modify a file, an add operation to inject a new file and a remove operation to delete a file. However, most current peer-to-peer storage applications are limited to file sharing: they do not implement the write and the delete operations.

Implementing the lookup and add operations is challenging in a peer-to-peer storage application because it does not provide a centralized host that associates file names with their corresponding locations: the file name database is distributed among the clients. Implementing the read operation is also challenging because it must scale with the number of readers.

Two applications are presented in this chapter: BitTorrent and Gnutella. BitTorrent only provides the read operations but it proposes an efficient algorithm that scales with the number of readers. Gnutella provides the add and lookup operations. The first version of Gnutella relies on a gossip protocol while the second version is semi-distributed: some clients are connected through a gossip protocol and others are connected through a classical client/server architecture.

Chapter written by Olivier MARIN, Sébastien MONNET and Gaël THOMAS.

4.2. BitTorrent

BitTorrent [COH 11, BIT 11] is a peer-to-peer file-sharing protocol that scales with the number of clients. It is specifically designed to handle large files such as movies or bulky software – for instance, linux distributions and games. The main idea behind BitTorrent is: "divide and conquer". As soon as a client has downloaded part of a file, it becomes a new server for this part. Other clients can then download this piece directly from this client. This mechanism avoids the bottleneck induced by a centralized server.

Studying BitTorrent is particularly interesting because it is one of the most efficient communication protocols enabling large files to be broadcast. This mainly explains its success: a study undertaken in February 2009 [SCH 09] estimates that BitTorrent generates between 27% and 55% of all Internet traffic – this value depends on geographical location.

The BitTorrent protocol relies on three main principles in order to ensure its performances [LEG 06]:

– tit-for-tat: a peer sends data to another peer only if the latter has given enough in exchange. This principle ensures the fairness of the BitTorrent network;

– priority to rare pieces: a peer downloads the rare pieces of the file first; rare pieces are those that are hosted only on a small subset of the peers. This principle ensures that even in the presence of churn (frequent connections/disconnections), all the pieces of the file are present in the BitTorrent network;

– optimistic unchoke: the tit-for-tat principle prevents a new peer which has nothing to share, from downloading its first pieces. The optimistic unchoke principle counterbalances the tit-for-tat principle by periodically selecting a random peer and sending it data freely.

4.2.1. *History*

The ancestor of BitTorrent is MojoNation, a communication protocol created by the "Evil Geniuses for a Better Tomorrow" company founded by Jim McCoy in 2000. MojoNation proposed to split a file into small pieces to avoid the bottleneck of a centralized server. The idea of tit-for-tat was already there, but with virtual money: each peer increased its credit by transferring requests of other peers. However, transferring the money degraded the performance as it requires authentication and encryption. The company went bankrupt in 2002. Bram Cohen left MojoNation in 2001 and began the development of BitTorrent. The first BitTorrent specification was presented at the CodeCon conference in 2002 in San Francisco [COH 02].

Until 2004, Bram Cohen's main income was based on donations collected through his web site (`http://bittorrent.org`), which provides the specification and free

implementation of BitTorrent. In 2004, he founded the commercial company called BitTorrent, Inc. (`http://bittorrent.com`) with his brother Ross Cohen and a business partner Ashwin Navin. BitTorrent, Inc. sells its services to movie studios.

One of the main original aspects of the BitTorrent specification is that its evolution has never broken the ascendant compatibility. This strength comes from the specification, which does not provide any algorithm as to the behavior of the peers but only defines their intention. Of course, the communication protocol (i.e. the formats of the messages) is fixed and cannot evolve. However, the way a peer takes a decision with respect to the information that it collects about the network is unspecified and varies from one implementation to another. Research is still being undertaken to improve the quality of the algorithms used by the peers.

4.2.2. *Terminology*

BitTorrent introduces a specific terminology to describe the architecture of the network:

– the swarm: the set of all the peers that are downloading or sharing a file;

– a piece: a piece of a file;

– the tracker: a centralized server that has a rough knowledge of the swarm. It knows the peers of the swarm and for every one of them, the pieces they host, which is only used for statistical purposes;

– a client: a peer that does not have all the pieces of the file. A client is therefore a peer that is trying to download new pieces of the file;

– a seeder: a peer that owns all the pieces of the file. When a client completes its download, it becomes a seeder;

– a leecher: a client that is only downloading. Most of the time, a leecher is simply a client that does not have any piece to share. However, a leecher could also be a selfish client.

In this section and the next, we reserve the term “peer” to name a client or a seeder.

4.2.3. *Joining a BitTorrent network*

When a peer wants to join a BitTorrent network, it contacts the tracker of the file. Finding the tracker is out of the scope of the protocol because BitTorrent does not address the problem of file lookup. Most of the time, classical HTTP servers maintain an association between file names and trackers. When the tracker receives a request from an incoming peer, it replies with a list of network peers (generally about 50).

Figure 4.1 illustrates the mechanism. Initially, seeder S_1 contacts the tracker. The tracker sends an empty list to S_1, but is now aware of the existence of S_1 and adds it to its own list. Afterwards, when C_1 contacts the tracker, it responds with its new list containing only S_1 and adds C_1 locally. Finally, as C_2 contacts the tracker it responds with the list $\{S_1, C_1\}$. In Figure 4.1, C_1 and C_2 already have some pieces of the file, this can happen when a client reconnects to the network after a disconnection.

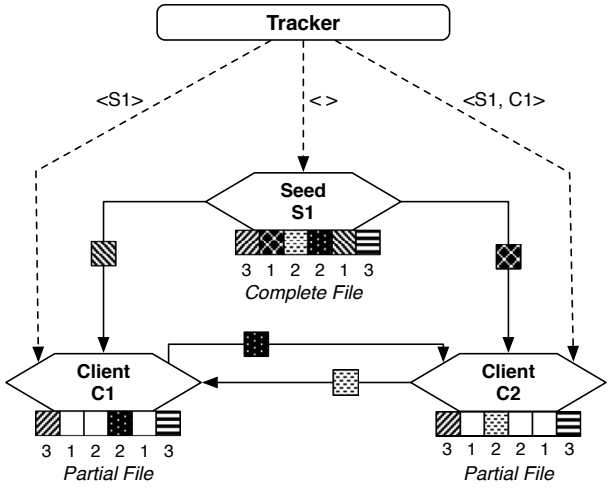

Figure 4.1. *Exchange of pieces*

4.2.4. *Making the pieces available*

Once a peer has acquired a list of neighbors from the tracker, it contacts all of them. A connection is bidirectional: when peers $N1$ and $N2$ are connected, they can exchange pieces of the file in both directions. In Figure 4.1, the graph is complete because C_1, S_1 and S_2 are all connected to one another. Such is generally not the case: neighborhoods do not span over the entire network as several thousands of peers can participate in the network while each peer only has around 50 neighbors.

As soon as two peers are connected, they exchange their list of available pieces. This list is used to determine which peer has which piece, but also to determine which pieces are sparse on the network. As the graph is not complete, a peer only estimates the availability; statistically, this estimation is correct with 50 peers. In Figure 4.1, the estimation corresponds to the real availability because the graph is complete.

When a peer downloads a new piece, it broadcasts this information to its neighbors and they update their availability lists.

A peer maintains three pieces of information about each of its neighbors:

– interested: if a peer C_1 is interested in one piece of C_2, then C_1 notifies C_2 and C_2 registers that C_1 is interested, otherwise, it registers that C_1 is not interested;

– choked: if a peer C_1 is interested in one piece of a peer C_2 and if C_2 cannot send data to C_1 because it already sends pieces to other neighbors, then C_2 registers that C_2 is chocked and notifies it. When C_2 sends data to C_1, it marks C_1 as unchoked and notifies it;

– snubbed: if a peer C_2 optimistically sends a piece to C_1, then C_1 is marked as snubbed for a short duration (see next section).

Figure 4.2 presents the state of two clients C_1 and C_2. C_2 is interested by a piece of C_1 (the third piece) and C_1 is not interested by any piece of C_2.

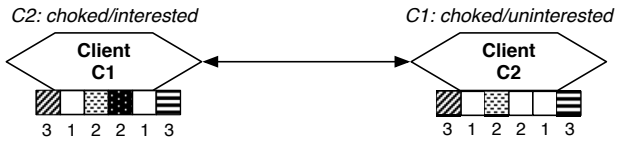

Figure 4.2. *State of two neighbors*

4.2.5. *Priority to rare pieces*

A peer will first try to download the rarest pieces offered by its neighbors. By minimizing the number of rare pieces, BitTorrent uniformly distributes the load on the peer and, therefore, increases the performances. In Figure 4.1, C_1 will therefore download the second piece from S_1 because it is the rarest piece offered by S_1. For the same reason, C_1 will also download the third piece from C_1. Symmetrically, C_2 will download the fourth piece from C_1 and the last piece from S_1 (the piece is chosen randomly among the rarest pieces, see below).

Giving a higher priority to rare pieces has a drawback. At some point, many clients can start downloading the same piece considered as the rarest. The result is counter-productive because these clients may therefore be unable to exchange other pieces amongst themselves. For example, on the Figure 4.1, the protocol must try to avoid the case where C_1 and C_2 download the same piece. To counterbalance this effect, a client randomly choses one of the pieces among the N rarest pieces where N is equal to the number of active connections (i.e. number of neighbors).

Distributing the rarest piece first increases the performance of a BitTorrent network but also has another advantage: as the seeders uniformly distribute their pieces, their

presence is requested for a short time. For example, after the exchange represented in the Figure 4.1, C_1 and C_2 can rebuild all the files even in the absence of the sender S_1.

4.2.6. *The tit-for-tat principle*

Neighbors exchange pieces by using a *tit-for-tat* algorithm: the more data a peer sends to its neighbors, the more data are sent by its neighbors. This algorithm forces the peer to send data: according to the tit-for-tat principle if a leecher only wants to download without sharing, it will not be serviced by the other peers.

To avoid overloading a peer, a peer only sends pieces to a small subset of its neighbors (normally four), other neighbors are just waiting (they are choked). To ensure the tit-for-tat algorithm, a peer choses its best neighbors and sends them pieces of the file.

The BitTorrent protocol uses one single criteria to find its best neighbors: *the download rate*. The download rate is equal to the number of bytes received by a neighbor plus one divided by the number of bytes received by this neighbor plus one. A client, i.e. a peer that does not have all the pieces, elects its best neighbors by choosing the neighbors with the highest download rates.

However, this criteria is useless for a seeder as it does not receive data. The goal of a seeder is different: it will try to give its piece as fast as possible. As a consequence, the more a neighbor downloads, the more it is interesting. A seeder therefore elects its neighbors with the smallest download rate. A client with a high network rate is favored because it can send its pieces faster.

As soon as a peer sends data to one of its neighbors, the download rate of this neighbor decrease. If two neighbors have approximately the same download rate and if the peer can only serve one if them, a fibrillation phenomenon could appear because the best neighbor will quickly become the worst. To avoid this phenomenon, the protocol defines rounds. A peer choices its best neighbors at the beginning of a round and does not change them if their download rates change during a complete round. By default, a round is 10 seconds long. Notice that a round is too short to receive a complete piece and that only smaller data, called data blocks, are exchanged. However, a peer can choke a client during a round if a best neighbor is not elected at the beginning of the round because it is not interested. In this case, if the neighbor becomes interested, the worst unchoked neighbor is choked to unchoke the best neighbor.

4.2.7. *The optimistic unchoke principle*

However, applying the tit-for-tat algorithm raises two problems:

– a client without any pieces will never obtain a piece if all its neighbors are clients;

– only the clients that have a lot of pieces will be elected as they can send a lot of data.

To solve this problem, a client reserves a slot to send data to a randomly elected neighbor. We say that the neighbor is optimistically unchoked. The optimistic unchoke algorithm is in contradiction to the tit-for-tat principle and an experience from 2006 [LOC 06] shows that a leecher can download a file without sending any pieces. However, it takes four time longer to download the file. By default, a client chooses optimistically a peer to unchoke every three rounds. At this round, if the client has four slots to send data, it chooses the three best neighbors and chooses the last one randomly.

When a client is served because it was optimistically unchoked, it is then marked as *snubbed* for a small period. When a client is snubbed it can not be elected by the optimistic unchoke algorithm.

4.2.8. *The messages of the protocol*

A message begins with a sequence of 4 bytes that gives the length of the message. Then, the message contains the type of message and possibly its content:

– `choked` (0): message of length 1 sent by a peer to inform one of its neighbors that it is choked and will not receive any data. This message follows an election;

– `unchoked` (1): message of length 1 sent by a peer to inform one of its neighbors that it is unchoked and will receive data. This message follows an election;

– `interested` (2): message of length 1 sent by a peer to inform one of its neighbors that it is interested in one of its pieces. This message follows a `bitfield` or a `have` message;

– `not interested` (3): message of length 1 sent by a peer to inform one of its neighbors that it is not interested in any of its pieces. This message follows a `bitfield` or a `piece` message from another peer;

– `have` (4): message of length 5 sent by a peer to inform its neighbors that it has the piece number encoded by the last four bytes. This message follows a `piece` message when a peer has a new available piece;

– `bitfield` (5) : message of length 1+N containing the list of its available pieces that a peer sends to its neighbors. This message is only sent when two peers meet for the first time. When a peer completes a new piece, it sends the lighter message `have` t oits neighbors;

– `request` (6): message of length 1+(2x4)+4 sent by a peer to request a given data block (number of the piece, number of the block, length). This message follows an `unchoke` message;

– `piece` (7): message of length 1+(2x4)+N that contains a data block (number of the piece, number of the block, data block). This message follows a `request`;

– `cancel` (8): message of length 1+(2x4)+4 sent to cancel a request. This message avoids a `piece` message when a peer receives the block from another peer.

4.2.9. *Conclusion*

The BitTorrent protocol relies on the tit-for-tat principle coupled with a priority given to rare pieces and the optimistic unchoke algorithm. The BitTorrent protocol optimizes the global state of the network while each peer only constructs a partial view of the network.

BitTorrent has a centralized point: the tracker. If the tracker fails, the BitTorrent network will disappear. Some solutions are proposed to avoid this centralized point. The simplest is the multi-tracker [HOF 11] which defines more than one tracker in the BitTorrent description file. Another way by using a DHT based on Kademlia [LOE 11]. The DHT does not change the communication protocol between peers but distributes the tracker: all peers of the swarm are connected through a DHT that lists all the peers.

4.3. Gnutella

Gnutella has remained one of the most popular peer-to-peer file sharing protocols since its creation in 2001. By the end of 2005, the Gnutella network comprized up to around $1,300,000$ nodes [STU 08], accounting for a 400% growth over the previous year. Since the appearance of the Kademlia network [KAD 11], there has been a decrease in gnutella network with users changing to Kademlia. Gnutella is a good example of a very large-scale network; it shows how such networks can be built.

The Gnutella protocol is semi-centralized, and relies on an *unstructured overlay* (see section 3.2.2). Two categories of nodes may be distinguished: ultra nodes (or ultrapeers) and leaf nodes. Contrary to classical semi-centralized network such as eDonkey [KLI 11], Gnutella nodes can be promoted to ultrapeer status or demoted to leaf status dynamically, in an effort to optimize performances on the fly as the network load evolves. Hence Gnutella is not semi-centralized in the same sense as its competitors, since it does not require a fixed infrastructure with predetermined central servers. Ultrapeers that form the core of the Gnutella network are actually promoted end-user nodes. They are interconnected on the basis of an unstructured overlay architecture, and leaf nodes become connected to the network by maintaining client/server links with ultrapeers.

There are several drawbacks to this architecture compared with other semi-centralized networks. The main one is that file lookups cannot be exhaustive, and are achieved

by flooding, this is presented in section 3.2.2. Another important drawback is that Gnutella is very sensitive to *churn* (see Chapter 3): as its core is composed of end-user nodes, it is hence highly volatile. Indeed, end-user nodes may enter and leave the network at any time. The Gnutella protocol must therefore be fault tolerant, thus preventing the untimely insertion/removal of nodes from either disturbing other nodes, or partitioning the network – and possibly destroying it.

4.3.1. *History*

Gnutella was initially designed by Tom Pepper and Justin Frankel in 2000 as a side project at NullSoft, Inc. The first prototype was only deployed for a day, on March 14, 2001 [GNU 11a], before the project was abandoned by AOL, which was then owner of NullSoft, Inc. However, even in such a short span, several thousand downloads were carried out: enough to establish a stable peer-to-peer network. After a few days of reverse engineering by third parties, the first Gnutella clones started appearing, and version 0.4 – the first official version of the Gnutella protocol – was defined.

Gnutella offered an alternative to centralized file-sharing solution such as Napster, and to semi-centralized ones, such as FastTrack (KazaA). The Gnutella protocol v0.4 became popular with the disappearance of Napster for legal reasons: Gnutella does not require any fixed infrastructure integrating central servers.

Version 0.4 of the Gnutella protocol assumed a fully decentralized network based on an unstructured overlay (see Chapter 3) in which all nodes are equivalent. This original version quickly demonstrated the limits of a such an architecture in terms of scalability, as the generated traffic increased exponentially with respect to the number of nodes. As a consequence, the Gnutella protocol evolved in 2002 to version 0.6 and has not been upgraded since. This newer version reintroduces the notion of ultrapeer used by semi-centralized networks. However, instead of requiring a fixed infrastructure based on central servers, ultrapeers are elected among end-user nodes.

One of the original aspects of the Gnutella protocol is that it is defined by the developers of the client software. No new version of the protocol has been formally specified since version 0.4: version 0.6 has no official specification [1]. The latter is still in use nowadays, and it is implemented by all clients of the network.

Gnutella version 0.6 proposes GGEP (Gnutella Generic Extension Protocol) which enables arbitrary extensions through messages extension blocks. Clients are allowed to discard the data encapsulated in these blocks, and thus vertical compatibility is maintained. However, no common consensus has been reached about what these

1. An RFC draft was written by the Gnutella community for version 0.6, but no final document was produced as Gnutella2 started appearing at the same period.

blocks should be used for. Yet clever usages have been implemented by most client applications, and hence the protocol keeps evolving without requiring any formal specification.

The Gnutella2 protocol was designed by Michael Stockes in 2002. It is not a successor to the Gnutella protocol being presented here, and both protocols are completely independent; they only share one type of message, and of course their name. There is a lot of disagreement in the Gnutella community over the matter of which protocol is the best between Gnutella v0.6 and Gnutella2. Both protocols address the scalability issues of v0.4, and most client applications provide the ability to participate to both networks simultaneously. This section is dedicated to a presentation of v0.6 rather than Gnutella2, as studying the former underlines more clearly which compromises were reached to overcome the limitations of v0.4.

4.3.2. *Architecture of the Gnutella v0.4 network*

Gnutella v0.6 heavily relies on the design of v0.4 [GNU 11b]. The aim of the present section is to describe this design: it is simpler than v0.6. The overlay, routing, and lookup aspects, before delving into the improvements brought by v0.6 will be the focus of the following section.

Any node in the Gnutella network is called a *servent* – which is a contraction of server and client. A servent is sometimes designated as client; this formulation is inadequate, however, as a servent does not just connect to a server but plugs itself directly into the network. There are many servent implementations, among which some popular programs are: LimeWire [LIM 11] (Java, OS independent), Shareaza [SHA 11] (Windows) and Gtk-gnutella [GTK 11] (Unix/MacOS).

Neighborhood

Every servent is connected to the network through its neighbors. Figure 4.3 illustrates a Gnutella network, where each servent has exactly two neighbors. Notice that, although a node s may consider another node t as its neighbor, t does not have to reciprocate this relationship. In this example, D is considered as a neighbor by nodes B, C, E, and F; yet D limits its set of neighbors to F and G. A node maintains a set of neighbors called its *neighborhood*.

More formally speaking, Gnutella's unstructured overlay is a directed graph where nodes are servents and arcs represent neighborhood relationships. The following three properties must be maintained in order for the network to remain optimally efficient.

PROPERTY 4.1 *The graph must be strongly connected.*

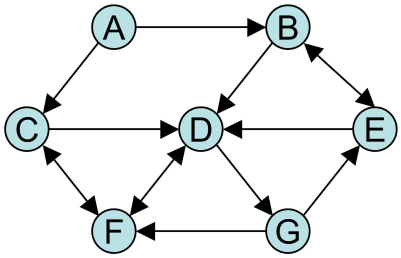

Figure 4.3. *A Gnutella neighborhood example*

Property 4.1 means that, given any two servents X and Y in the network, there must be at least one path leading from X to Y. This property is fundamental in order to avoid a partitioning of the Gnutella network into several disconnected subnetworks.

PROPERTY 4.2 *Neighborhoods must be distributed uniformly.*

Property 4.2 actually states that, if n is the average number of neighborhoods every servent belongs to, then the standard deviation around n must remain low. This property guarantees that no servent will be submitted to excessive workloads. In the example given in Figure 4.3, D is considered as a neighbor by four other nodes; this is significantly above the average value of two.

PROPERTY 4.3 *The neighborhood set of a servent must never be empty.*

A node with an empty neighborhood set will eventually be expelled from the network.

An elegant technique maximizes the probability of guaranteeing properties 4.1 and 4.2: every node selects its neighbors randomly. Random selection ensures that neighborhoods are distributed uniformly and prevents the formation of weakly connected subgraphs. In order to maintain property 4.3, a threshhold value is associated with the size of the neighborhood set, below which new neighbors have to be sought. The Gnutella v0.4 specification recommends a threshhold value of 5.

In a nutshell, Gnutella nodes strive to maintain all three properties by selecting neighbors randomly and by keeping track of at least five neighbors.

Routing and overlay

Gnutella provides query mechanisms to look up files according to multiple criteria: keywords, file size, or file type. It also enables the search for new neighbors when a node's lower threshold on its neighborhood size is reached.

The Gnutella protocol is based on an unstructured overlay and neighbors are selected randomly. Using this class of overlays, a search operation is based on a *flooding* algorithm (see Chapter 3). Hence, in order to maximize its success, a file search must be broadcast to as many nodes as possible. The higher the number of nodes queried, the greater the chance of getting a positive reply from one of them.

To initiate a search, a servent emits a request to all its neighbors. The initiator of a query is the servent that instigates a flood through the network. Receiving a request the neighbor nodes will in turn retransmit the query to their own neighbors. Flooding algorithms raise two major issues:

– a query must terminate eventually, (i.e. it cannot be retransmitted forever throughout the network). For this purpose, a *time-to-live* (TTL) counter is associated with every request message. This counter is decremented at every hop, and the request stops being forwarded when the counter value reaches zero;

– cycles, where the same nodes forward the same message among themselves, must be avoided. This is acheived by associating a unique identifier to every request. A servent will forward a request it receives for the first time, but discards it when it receives it for a second time.

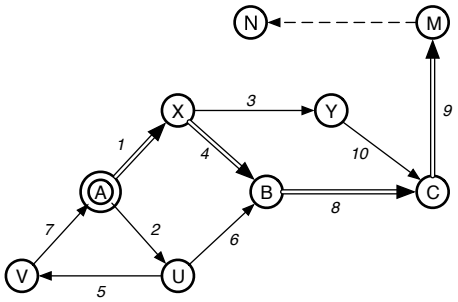

Figure 4.4. *Gnutella network routing*

Figure 4.4 illustrates how Gnutella routes requests. The numbers on the arc labels represent their order of transmission. Servent A instigates the search with a TTL value of 4. Servent N will never be aware of this particular search as it is too far from A with respect to the associated TTL. When A, B and C receive the request for the second

time – through messages 7, 6, and 8, respectively – discard it in order to avoid routing through the same paths.

With Gnutella, search instigators remain anonymous as a request does not retain the address of its originating node. Ethically speaking, this mechanism for preserving the anonymity of query initiators is disputable, as it makes it absolutely impossible to ascertain which servent is looking for the given data. This approach prevents a servent from responding directly to the instigator of a request. An other drawback is that this forestalls the identification by law enforcement agents of individuals looking for illicit content, such as illegal copies of movies or outlawed pornographic data.

A naïve approach would be to respond to a request by flooding. This solution is both inefficient and impractical as neighborhood status is asymmetric. The initial TTL value of the request may then be insufficient for the reply to reach the query instigator, this is the case in Figure 4.4 where node U can be reached from A in one hop, but it requires two hops for an answer.

Gnutella provides a more efficient solution. Every time a node receives a request, it extracts the identity of the sender from the message, logs it locally, and replaces it by its own identity before forwarding the message. Thus a routing path to the query instigator is maintained as a request message is being propagated. A response is sent by passing a reply message along this path. In the example shown in bold in Figure 4.4, node M can respond to A through route {M, C, B, X, A}. Thus a request floods the network, while the routing tables get built so as to convey the response to the query instigator.

Bootstrapping and acquiring new neighbors

One of the main issues that must be addressed by Gnutella is the search for new neighbors. Indeed, servents join and leave the network continually – a phenomenon referred to as *churn* – and Gnutella avoids server nodes with predetermined addresses.

The lifecycle of a servent comprises three states:

– an *initial state*, in which the servent knows no other node in the network;

– a *startup state*, in which the servent knows the addresses of other network nodes but is not yet linked to them;

– a *connected state*, in which the servent is linked to at least one neighbor.

Every servent maintains a cache of potential neighbors, of which only a subset constitutes the neighborhood set. Elements of the latter set are nodes to which the servent is actually connected. The servent is not connected to any of the remaining nodes in the cache, and has no clue with regards to their liveness.

Figure 4.5 recapitulates the possible states of a servent. A servent program starts in the initial state if its cache is empty, in the startup state otherwise. As soon as it has achieved the connected state, a servent participates in the network and can start launching queries. Obviously, the aim of the Gnutella protocol is to prevent servents from falling out of the connected state.

Switching from the initial state to the startup state is called the *bootstrap*. The Gnutella protocol does not handle this phase; instead, most servents use a secondary protocol called GWebCache [DÄM 03]. A GWebCache server is actually an HTTP server that maintains a cache of live servents connected to the network. Once connected, a servent registers on a GWebCache server. A servent in the initial state may download the contents of a GWebCache and copy them to its own cache [2]. This approach automates the bootstrap, but requires a fixed infrastructure. To the best of our knowledge, the only possible technique for acquiring new neighbors in a dynamic infrastructure is to exchange addresses among friendly acquaintances through alternative means – IRC, mail, or even over the phone.

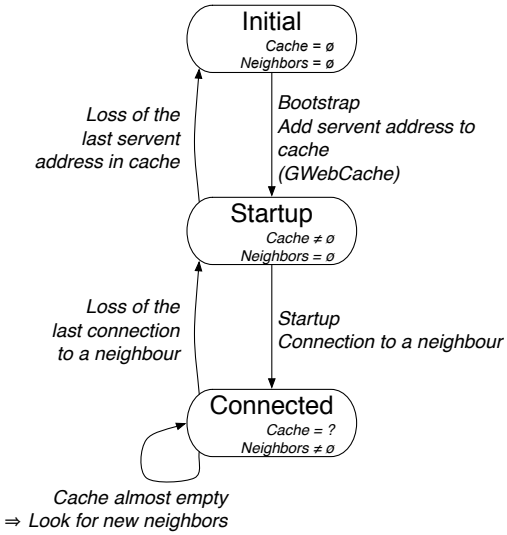

Figure 4.5. *Servent initialization*

Switching from the startup state to the connected state is called the *startup*. A servent randomly selects nodes out of its cache and tries to connect to them. Those

2. To do so, the servent has to know at least one GWebCache server address.

that have left the network and cause the connection to fail are deleted from the cache. As soon as a connection is achieved, the servent switches to its connected state.

In the connected state, a servent strives to sustain its neighborhood set so that its size remains above the threshhold value. For this purpose, it periodically connects to new servents in its cache. The size of the neighborhood set may decrease when a neighbor is assumed to be disconnected: this occurs when a timeout associated with the connection expires.

Obviously, this failure-detection scheme is very basic and not really tailored to an asynchronous network such as the Internet [FIS 85]. The suspected servent may be too overloaded to demonstrate its own liveness, yet still alive. The problem may also come from the network, too overloaded itself to convey messages on time. However, mistaken failure detections have almost no impact on a Gnutella network as all nodes are equivalent. Any node may be replaced by another, and the propagation of messages by flooding ensures that some of these can be lost at very low cost.

In order to avoid the profusion of obsolete addresses in a servent cache, the servent adds every new node address it discovers in received messages. Conversely, to avoid overstretching the cache, only the most recently used addresses are kept.

A servent that is either inactive or offline may not be able to update its cache for a long period of time. When the number of live nodes in its cache becomes too small, it launches a neighbor search request. Receiving such a request, a servent sends its own cache list as a reply. Thus, flooding the network with a neighbor lookup request enables the communication of several cache lists simultaneously. Every node along the path of such a request will benefit from it, as it also updates its own cache with the contents it forwards. This scheme enables servent caches to be updated in a fully distributed way, and preserves GWebCache servers from overload.

4.3.3. *Implementing Gnutella 0.4*

The Gnutella protocol specifies five different message types:

– *Ping*: a neighbor search request;

– *Pong*: the answer to a *Ping*;

– *Query*: a file search request;

– *QueryHit*: the answer to a *Query* sent by a servent when it knows the answer;

– *Push*: a control inversion scheme used by a sender and a receiver in order to bypass a firewall.

Ping and *Query* messages are propagated by flooding. The others are answers: a *Pong* replies to a *Ping*, a *QueryHit* replies to a *Query* and a *Push* also replies to a

QueryHit in some circumstances. Answers are conveyed along the path built during the flooding.

All messages used over the Gnutella 0.4 network display the same header, which enables the routing mechanism. The composition of the header is as follows:

– a unique request identifier, used build the routing table and to prevent cycles during a flooding;

– the TTL of a message, decremented at every hop until it reaches zero, where the message is discarded;

– the number of hops already achieved, incremented at every hop and used to set the TTL of a reply;

– the message type: *Ping*, *Pong*, *Query*, *QueryHit* or *Push*;

– the message size.

File transfer

File download/upload is a feature that is independent of the Gnutella network, in the sense that two servents wishing to exchange a file – one of the servents shares it, the other one downloads it – do so directly without involving any other node. This implies that anonymity cannot be fully enforced in Gnutella, as any servent may find out the location of nodes that share a given file.

Let l be the servent that initiates a file search by emitting an original *Query*. The search eventually terminates, at which time l acquires a list of couples (F, O), where F is a file name and O is the address of a node that shares F[3]. l can distinguish files that are shared at multiple locations: they are the files with the exact same F value. l then associates a list of servents O that share it to every F, and displays all the different values of F that were returned in response to the query to the end user. Once the end user has selected one of the F values, l attempts to download the file from the sharer nodes it knows of. The transfer itself is made by *chunks*: the file is split in equally sized parts, and every part can be downloaded from a different owner.

Every servent provides a small HTTP server with the ability to interpret a specific `GET` command: `GET /get/`*`index/nom`*` HTTP/1.0\n˚`

```
Connection: Keep-Alive\n\r
Range: bytes=0..n\n\r
User-Agent: Gnutella\n\r
```

The only original aspect of the file transfer protocol is the `Range` parameter, which specifies the size of the chunks in order to allow multiple simultaneous downloads.

3. O stands for owner.

As soon as it has successfully downloaded an entire chunk of F, l itself becomes an owner of the file and may send positive replies to queries regarding F. Of course, it can only serve the chunks of F that it has already downloaded. Similarly to BitTorrent, downloads become parallelized very quickly.

Limitations of the Gnutella v0.4 protocol

In order to associate requests with their responses – a *Ping* with a *Pong*, a *Query* with a *QueryHit* and a *QueryHit* with a *Pong* – a servent must log all received messages. However, its memory capacity is limited; thus saved data must have a finite lifespan. The lifespan value depends heavily on the traffic (the number of received requests per time unit), the transit time of a request, and the available memory on the servent.

When a servent memory becomes saturated, there are two alternatives: (i) delete logged requests faster or (ii) let the memory saturate entirely. Neither of these solutions is satisfactory.

The first solution causes a significant increase in traffic as less requests get discarded for flooding when they are received more than once on the same node: this saturates the bandwidth instead of the memory. Moreover, if a node has deleted the address of the sender it received a request from, it can no longer convey the response back to the query instigator. This in turn may prompt the instigator to re-emit its query, and will lead to an even more dramatic saturation of the network.

The second approach, a full saturation of the memory, causes the servent to significantly slow down. Therefore, the forwarding delay increases for every message transiting on the node, as processing data takes more time. The direct consequence is that the neighbors of the servent whose memory is saturated will be required to log messages for longer durations, and hence risk memory saturation themselves.

This snowball effect caused by epidemic memory saturation was indeed observed on the Gnutella v0.4 network [RIT 01] when low-speed dial-up connections were still common.

In order to solve this problem, developers reintroduced the notion of super-peer already in use in semi-structured networks. Super-peers are called *UltraPeers* in the Gnutella network.

4.3.4. *Evolution to Gnutella protocol v0.6*

Version 0.6 of the protocol takes into account the aforementioned heterogenity of nodes in terms of capacity and bandwidth. In order to prevent the entire network from being handicapped by the slowest nodes, servents are separated into two categories:

– ultraPeers: stable nodes with a high bandwidth;
– leaves: all the other nodes, bound to become ultraPeer clients.

UltraPeers constitute the core of a Gnutella network and use its original communication protocol: queries are sent by flooding and results are conveyed along the request paths. Leaves, however, are simple clients of the ultraPeers. The network core thus groups the most efficient servents and is more homogeneous, hiding the limitations of the original protocol.

This architecture breaks the symmetry among servents, and becomes similar to the eDonkey network, where the core is a peer-to-peer overlay and clients are connected to the core through a client/server connection. The main difference with eDonkey resides in the dynamicity of the core structure. eDonkey favours a static overlay where ultraPeers are clearly identified, whereas ultraPeer status is dynamic in Gnutella. Gnutella leaves can be promoted to ultraPeer status if they satisfy several eligibility criteria, and ultraPeers get demoted to leaf status when they stop satisfying them.

Both the Gnutella v0.6 architecture and protocol are very similar to that of v0.4. The following sections describe how some mechanisms may differ from the original version, and detail elements that were not mentioned in previous sections.

Network architecture and protocol mechanisms

Theoretically, ultraPeers have greater bandwidth then the average servent in v0.4, hence they can handle a higher number of neighbors. The typical utlraPeer is supposed to keep up permanent connections with 30 other ultraPeers, and to service between 30 and 45 leaves. A leaf aims to stay connected to three ultraPeers to ensure that it will not leave the network.

In essence the Gnutella v0.6 protocol works in a similar way to v0.4: the message structure remains the same. The fundamental difference is that only ultraPeers use the flooding algorithm. They keep track of all the files shared by their leaves and send responses in their stead. Figure 4.6 shows the scenario that unfolds when leaf L1 initiates a search for file xyz in a v0.6 network. L1 sends a *Query* message to its ultraPeers – U2 in this example. U2 floods the network core with this message and waits for the results. Upon receiving this request, U1 acts as a proxy for L2: it sends a *QueryHit* message back along the path to U2, which in turn forwards it to L1.

Ping and *Pong* messages in the network core are completely isolated from their matching counterparts in the periphery. In the example of Figure 4.6, when L3 sends a *Ping* message to its ultraPeer, U3 does not flood the core with this request and sends its own cache back directly. However, if the cache of an ultraPeer becomes too small, the node does flood the network core with its own *Ping* message and acts exactly as described in Gnutella v0.4.

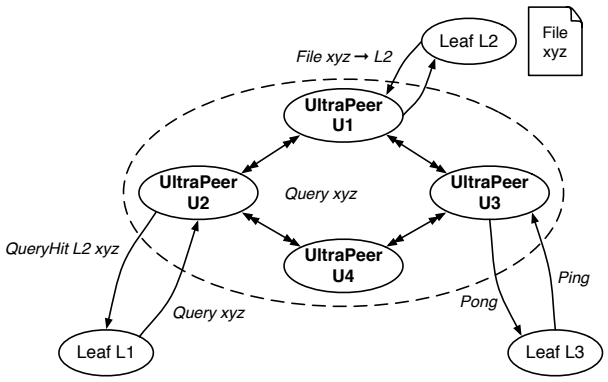

Figure 4.6. *File search example in Gnutella v0.6*

Although it is privileged in terms of access to the data, an ultraPeer remains a servent node with a user behind it. Therefore, an ultraPeer may initiate its own searches, in which case it behaves as if it were a leaf.

This architecture solves the limitations of the v0.4 network by avoiding low bandwidth servents along the message paths. It also prevents the saturation of leaf node resources that are usually more scarce.

Version 0.6 of the Gnutella protocol also offers several additional improvements, such as the ability to cancel a request once the initiator is satisfied with the results. This technique is particularly useful for files that are extremely popular, and get shared by a huge number of nodes.

UltraPeer election

To become an ultraPeer, a servent must satisfy the following properties:

– it must not live behind a firewall;

– it must execute an operating system able to open a large number of socket connections (Linux, Windows 2000/NT and greater, Max OS 10 and greater);

– it must have a good network rate (10 kb/s for download and 15 kb/s for upload);

– it must be stable, i.e. the servent must be connected for some hours;

– it must be powerful (good CPU, large memory).

When a servent has all these properties, it is "able to become an ultraPeer". The servent self-evaluates its ability to become an ultraPeer and nothing prevents a malicious servent from becoming an ultraPeer even if it does not satisfy all these properties.

The election of an ultraPeer implements a simple protocol which aims to stabilize their number. Technically, when a peer A meets a peer B and wants to add B to its neighborhood, different possibilities must be considered:

– **Case 1.** A and B are leaves:
- **1.a.** if B only implements Gnutella 0.4, it accepts the connection of B. B then uses A to find other peers;
- **1.b.** if B implements Gnutella 0.6. If B knows ultraPeers, it declines the connection and send the address of an ultraPeer to A. Otherwise, B asks A to become an ultraPeer. If A is able to become an ultraPeer, it becomes an ultraPeer, otherwise, B becomes a neighbor of A as in the protocol 0.4;

– **Case 2.** A is a leaf and B is an ultraPeer: B accepts the connection of A. However, if B has too many connections, it asks A to become an ultraPeer. If A is able to become an ultraPeer, A reconnects B as an ultraPeer (see case 4). In all cases, if A had neighbors as a leaf (see case 1), A sends its neighbors to B and they try to connect B;

– **Case 3.** A is an ultraPeer and B is a leaf: this case is impossible. If A is an ultraPeer it will never choose a leaf as a neighbor;

– **Case 4.** A and B are ultraPeers:
- **4.b.** if A estimates that it does not have enough leaves, it will ask for B some of its leaves. If B has no more leaves, it becomes a leaf of A.
- **4.b.** otherwise, A and B become ultraPeer neighbors.

To stabilize the number of ultraPeers, two mechanisms are used. A leaf becomes an ultraPeer when it connects an overloaded ultraPeer (case 2) or when it has neighbors (case 1) and becomes able to be an ultraPeer. An ultraPeer becomes again a leaf when one of its neighbors takes all its leaves (case 4). This simple algorithm balances the number of ultraPeers in the network without requiring a global knowledge: the balance is only local to a peer and its neighborhood, but, little by little, a balance between leaves and ultraPeers is respected all over the network.

To summarize, the Gnutella protocol is distributed and does not require a fixed infrastructure: Gnutella does not require clearly identified peers that never crash. The original version of the protocol shows how to build an entirely unstructured overlay where all peers are strictly equivalents. However, the deployment of this first version showed degradations of performances around weak nodes.

The solution brought by the version 0.6 enhances the algorithms by defining ultraPeers and leaves. The backbone of the Gnutella 0.6 network is defined by gossip with the ultraPeers. Each ultraPeer acts as a server for a small set of clients.

The main originality of this protocol is the election of ultraPeers: the overlay does not require a fixed infrastructure and only powerful and stable leaves can become ultraPeers. The algorithm stabilizes the number of ultraPeers without constructing a global view of the network. When an ultraPeer has too many leaves, one of the

leaves is promoted to become an ultraPeer and reciprocally. If two ultraPeers with few leaves meet, one of the ultraPeers becomes a leaf. Experimentally, this simple algorithm using a rough estimation globally stabilizes the number of ultraPeers.

4.4. Conclusion

Our study of BitTorrent and Gnutella shows how it is possible to build scalable distributed networks without relying on global knowledge. Each peer only has a partial view of the network and takes its decisions statistically. The algorithm that promotes rare pieces in BitTorrent or the algorithm that elects an ultraPeer in Gnutella only requires an estimation given by the neighborhood of a peer. Globally, these networks stabilize some properties: pieces of files are uniformly distributed in BitTorrent and the number of super-peers remains proportional to the number of leaves.

Studying and developing algorithms that scale with the size of the network remains a complex subject because, most of the time, only experimental observations can uncover the weaknesses or the strengths of these algorithms. For example, only the observation of the Gnutella 0.4 network can show that the presence of some weak peers can drastically degrade the performance ok a large portion of the network.

Both the applications studied in this chapter can intrinsically scale with the size of the network because the manipulated data are only read. If a peer wishes to modify a file it simply creates a new file that is in no way linked to the original. Building a scalable network that supports writable data also remains an open subject because different versions of the same file will exist in the network and these replicas must eventually converge to the same value.

4.5. Bibliography

[BIT 11] BITTORRENT, "Bittorrent protocol specification v1.0", http://wiki.theory.org/BitTorrentSpecification, 2011.

[COH 02] COHEN B., "BitTorrent - hosting large, popular files cheaply", *CodeCon*, 2002.

[COH 11] COHEN B., "The BitTorrent protocol specification", http://www.bittorrent.org/beps/bep_0003.html, 2011.

[DÄM 03] DÄMPFLING H., "Gnutella web caching system - version 2 specifications client developers' guide", http://www.gnucleus.com/gwebcache/newgwc.html, 2003.

[FIS 85] FISCHER M. J., LYNCH N. A., PATERSON M. S., "Impossibility of distributed consensus with one faulty process", *J. ACM*, vol. 32, p. 374–382, ACM, 1985.

[GNU 11a] GNUTELLA, "Gnutella announcement on SlashDot", http://slashdot.org/article.pl?sid=00/03/14/0949234, 2011.

[GNU 11b] GNUTELLA, "The Gnutella protocol specification v0.4 – document revision 1.2", www.stanford.edu/class/cs244b/gnutella_protocol_0.4.pdf, 2011.

[GTK 11] GTK-GNUTELLA, "Gtk-Gnutella home page", http://gtk-gnutella.sourceforge.net, 2011.

[HOF 11] HOFFMAN J., "Multitracker metadata extension", http://www.bittorrent.org/beps/bep_0012.html, 2011.

[KAD 11] KADEMLIA, "Kademlia: a design specification", http://xlattice.sourceforge.net/components/protocol/kademlia/specs.html, 2011.

[KLI 11] KLIMKIN A., "eDonkey protocol specification 0.6.2", http://sourceforge.net/projects/pdonkey/, 2011.

[LEG 06] LEGOUT A., URVOY-KELLER G., MICHIARDI P., "Rarest first and choke algorithms are enough", *Internet Measurement Conference*, p. 203-216, 2006.

[LIM 11] LIMEWIRE, "Lime Wire home page", http://www.limewire.com, 2011.

[LOC 06] LOCHER T., MOOR P., SCHMID S., WATTENHOFER R., "Free riding in BitTorrent is cheap", *5th Workshop on Hot Topics in Networks (HotNets), Irvine, California, USA*, November 2006.

[LOE 11] LOEWENSTERN A., "DHT protocol", http://www.bittorrent.org/beps/bep_0005.html, 2011.

[RIT 01] RITTER J., "Why Gnutella can't scale. No, really.", http://www.eecs.harvard.edu/ jonathan/papers/2001/ritter01gnutella-cant-scale.pdf, 2001.

[SCH 09] SCHULZE H., MOCHALSKI K., "Internet study 2008/2009", http://www.ipoque.com/userfiles/file/ipoque-Internet-Study-08-09.pdf, 2009.

[SHA 11] SHAREAZA, "Shareaza home page", http://www.shareaza.com, 2011.

[STU 08] STUTZBACH D., REJAIE R., SEN S., "Characterizing unstructured overlay topologies in modern P2P file-sharing systems", *IEEE/ACM Transactions on Networking*, vol. 16, p. 267-280, 2008.