MASTER'S THESIS

Gridella: an open and efficient Gnutella-compatible Peer-to-Peer System based on the P-Grid approach

Information Systems Institute
Distributed Systems Group
Technical University of Vienna

under the guidance of
ao.Univ.Prof. Dipl.-Ing. Dr.techn. Harald Gall
and
Dipl.-Ing. Dr.techn. Manfred Hauswirth
Laboratoire de Systèmes d'Information Répartis (LSIR)
École Polytechnique Fédérale de Lausanne (EPFL)
as the contributing advisor responsible

by

Roman Schmidt
Heinrichsgasse 2/4
1010 Wien
R.Schmidt@infosys.tuwien.ac.at
Matr.Nr.: 9625228

Vienna, October 2002

Kurzfassung

Diese Diplomarbeit beschreibt die Peer-to-Peer Applikation Gridella. Der dezentrale Aufbau von P2P Systemen erlaubt es jedem Knoten Suchabfragen alleinig mit lokalen Interaktionen zu beantworten. Das in dieser Arbeit beschriebene System skaliert für hohe Knotenzahlen und Datenmengen und bleibt auch trotz fehlerhafter Knoten verfügbar. Weiters können die Wahrscheinlichkeiten für erfolgreiche Suchabfragen angegeben werden.

Gridella beruht auf dem P-Grid Ansatz [1, 5] und ist kompatibel zu Gnutella ausgelegt, um die vorhandene Gnutella-Infrastuktur „infiltrieren“ zu können und damit eine einfache Migration zu ermöglichen. Durch P-Grid kann sowohl die Effizienz bei Suchabfragen gesteigert werden, als auch die erforderliche Bandbreite für die Systemerhaltung reduziert werden.

Nach einer Analyse der bestehenden Gnutella-Infrastruktur und deren Probleme, werden die Grundlagen von P-Grid präsentiert. Die entwickelte Applikation unterstützt sowohl das bestehende Gnutella- als auch das neue Gridella- Protokoll. Danach werden die Architektur und Kommunikation innerhalb des Systems ausführlich beschrieben.

Abstract

This thesis describes the Peer-to-Peer Application Gridella. The decentralized architecture of P2P systems enables each peer to fulfill search requests solely by local interactions. The system described in this thesis scales to high numbers of peers and data items and remains available in spite of failing peers. Further, probabilities for successful search requests can be given. Gridella is based on the P-Grid approach [1, 5] and is designed to be compatible with Gnutella " infiltrate" the existing Gnutella infrastructure and enable an simple migration. P-Grid increases the efficiency of search requests, and reduces the required bandwidth for system maintenance. On the basis of an analysis of the existing Gnutella infrastructure and its problems I describe the foundations of P-Grid. The developed application supports the existing Gnutella and the new Gridella protocol. Then the architecture and the communication between the systems are described in detail.

Acknowledgments

First of all I want to thank my mother Annemarie and my father Franz for their support during the last years. They have made it possible for me with there patience and financial support to finish my studies and this master thesis.

Secondly I want to thank Manfred Hauswirth for his support during the whole work. I am especially grateful for the many hours that he spent as advisor as well as reviewer for this thesis. I would also like to mention his engagement for me in some financial matters.

Further I want to thank the whole team around Manfred Hauswirth at the EPFL in Lausanne, including Karl Aberer and Magdalena Punceva, for their work on the mathematical foundations of P-Grid.

Last but not least I want to thank my fellow student and friend Hannes Stratil for his support and humor during the last years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The limitations of client/server-based systems become evident in an Internet-scale distributed environment: Resources are concentrated on a small number of (virtual) nodes which must apply sophisticated load-balancing and fault-tolerance algorithms to provide users with continuous and reliable access. Additionally, network bandwidth to/from successful Internet servers must be increased steadily because no "a priori" caching and replication strategies exist. These concepts were introduced "a posteriori" when the WWW as the most successful Internet service developed into a network bandwidth nightmare. A considerable amount of scientific work (HTTP, caching infrastructures) now goes into remedying these problems.

If the provider of a service wants to stay "in power" in terms of confidentiality of data and access control the client/server (C/S) approach is still the best one despite its problems. For several other application domains Peer-to-Peer (P2P) systems offer an alternative to traditional client/server systems: Every node (peer) of the system acts as a client and server (servent) and provides a (possibly replicated) part of the overall information available from the system. Each peer "pays" its participation by providing access to its computing resources. P2P systems can be characterized by the following properties:

- no central coordination
- no central database
- no peer has a global view of the system
- global behavior emerges from local interactions

- all existing data and services should be accessible
- peers are autonomous
- peers and connections are unreliable

Several P2P systems exist and have proven successful. This thesis presents a P2P system based on the P-Grid approach [1] which addresses several of the inefficiencies and problems of current systems.

1.1 Motivation

The P2P approach is intriguing because it circumvents many problems of C/S systems in a very simple fashion. However, it pays for this simplicity with considerably higher complexity for searching, node organization, security, etc. That is the reason why successful systems "neglected" such problems. For example, Napster [24], which made the P2P idea popular, in fact is a centralized database holding references to files on peers. This means that taken the above properties of a P2P system it is not strictly a P2P system. Gnutella [14] solves this and other problems, but at the cost of using a communication-intensive search mechanism. A considerable amount of research exists in the related areas of distributed and cooperative information systems which is not exploited by the P2P community yet. Still P2P systems are based on very simple approaches which are easy to deploy but suffer from problematic shortcomings.

Gnutalla is a very good example for such P2P systems. Its protocol is very simple and this leads to several problems. As it will be shown later in Chapter 2, its communication-intensive search mechanism leads to high bandwidth consumption for every peer. This has already caused problems, because peers with low speed connections could not keep up as traffic increased. Another (social) problem are the so-called free riders. They do not share files and resources but only produce search requests. If the number of free riders increases further, Gnutella be transformed from a P2P into a C/S-like system with a backbone structure. These servers could then be attacked easily by query-flooding or similar attacks. Gnutella also implies total trust in the network, because there are no mechanism to prove the identity of peers or the contents of shared files. Shared files with popular filenames could be misused to distribute a virus or a trojan to attack the downloading host.

These were some of the motivations for us to develop a more advanced P2P system in which we try to address all P2P properties and take into account recent research results from related areas. In this thesis I describe Gridella, our Gnutella compatible P2P system which is based on the P-Grid approach [1]. We wanted Gridella to be compatible with Gnutella to “infiltrate” the existing Gnutella infrastructure and gradually replace standard Gnutella nodes with Gridella nodes. This approach brings together the best of both worlds: Gridella can communicate with existing Gnutella nodes but can offer advanced services when communicating with other Gridella nodes. By using the Gnutella protocol [14] as one of Gridella's communication protocols we try to offer a simple and smooth migration path.

1.2 Goals

The main goal of this work is to develop the Gridella application with a modular architecture, which then can be used to construct the P-Grid infrastructure. Once the infrastructure is created, the software can be used to perform user-queries, handle the results and enable the user do download the found information.

Gridella supports the Gnutella protocol to interact with Gnutella. Additionally a special Gridella XML-based protocol is designed and implemented that supports Gridella's additional capabilities for efficient searches. The protocol used to communicate with other servents is chosen in the handshake process after establishing the connection. Gridella is the default protocol and Gnutella the alternative. Other P2P protocols can be added later easily because of Gridella's modular design.

Gridella has a state-of-the art graphical user interface (GUI) to present all its activities to the user in an adequate form.

1.3 Organization of the Thesis

The thesis is structured as follows: Chapter 2 defines the current problems and describes Gnutella's underlying architecture and communication patterns to understand its benefits and problems which motivated the development of a new P2P system. To provide a direct comparison with Gridella this is followed by a presentation of Gridella's essential algorithmic foundations: P-Grid (Section 3.1), the process of constructing the P-Grid (Section 3.2), and a mapping scheme for search keys (Section 3.3). Chapter 4 then provides the details of Gridella's architecture and interaction patterns and compares it with Gnutella. Chapter 5 presents the Gridella protocol, followed by the component design of the software in Chapter 6. Chapter 7 presents the implementation of the Gridella application. I relate Gridella to other approaches in the field in Chapter 8, present an evaluation (Section 9.1) and possible future work (Section 9.2) and give the conclusions in Chapter 10.

Chapter 2

Problem Statement

2.1 State-of-the-Art

The first popular P2P system was Napster [24] which could only be used to share mp3 and wma files. In fact it is not strictly a P2P system because it uses a central server to index all shared files. As this server is also necessary to locate files Napster could be shutdown easily by order of the court.
Audiogalaxy [7] and iMesh [19] are very similar to Napster and thus have the same problems.

The FastTrack [11] P2P stack is used by several applications such as KaZaA or Grokster to create a file sharing network. Its protocol is not open and all applications are licensed by FastTrack. They support a metadata search for any type of media files like audio, video, images, documents, and software. If possible, this information are derived automatically from the file, or entered by the user via the application's file import wizard. While FastTrack is largely a decentralized system, the speed of its query engine rivals that of centralized systems like Napster. A central server is still responsible for maintaining user registrations, logging users into the system (in order to maintain active user statistics, etc.), and bootstrapping the peer discovery process [29]. After a peer is authenticated to the server, the server provides IP addresses and ports (always 1214) of one or more "SuperNodes" to which the peer then connects. A SuperNode acts like a local search hub, building an index of the media files being shared by each peer connected to it, and proxying search requests on behalf of these peers. SuperNodes are connected among themselves to forward search requests of connected peers. This scheme as shown in Figure 2.1 greatly reduces search times in comparison to a broadcast query algorithm like that employed on the Gnutella network. Peers are elected automatically to become SuperNodes if they have sufficient bandwidth and processing power (a configuration parameter allows users to opt out of running their peer in this mode). Search results contain the IP addresses of peers sharing the files that match the search criteria, and file downloads are done via the HTTP protocol in a peer-to-peer way [29].

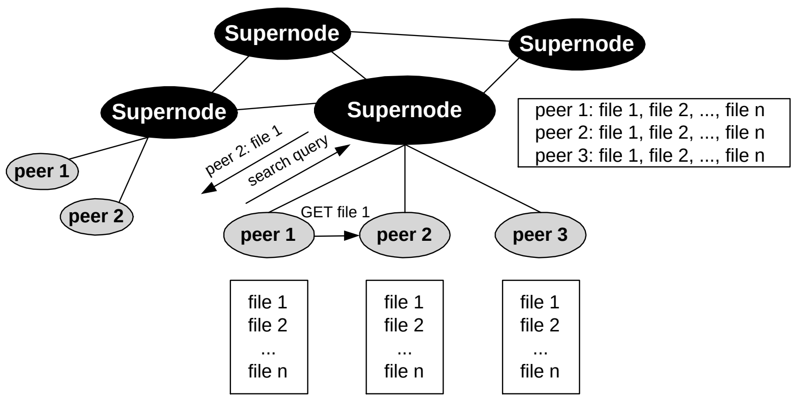


Figure 2.1: FastTrack architecture with SuperNodes

Two interesting features of FastTrack are its so-called SmartStream and FastStream technologies. SmartStream implements a type of fail-over system that attempts to locate another peer sharing the same file, and automatically tries to resume an incomplete download. FastStream is intended to solve another issues of primary concern in P2P file-sharing systems: slow downloads. When the search engine finds that more than one active peer is serving a particular file, the download task is shared among these peers to improve the download performance.

2.2 Gnutella

The Gnutella system is a decentralized file-sharing system whose participants form a virtual network communicating in a P2P way via the Gnutella protocol [9]. The Gnutella protocol is a simple protocol for distributed file search. To participate in Gnutella a peer first must connect to a known Gnutella host (servers such as *gnutellahosts.com:6346* return lists of hosts to get started; this is outside the Gnutella protocol specification). The protocol consists of 5 basic message types shown in Table 2.1.

Type	**Description**	**Contained Information**
Ping	Announce availability and probe for other servents	None
Pong	Response to a Ping	IP address and port number of the responding servent; number and total kB of files shared
Query	Search request	Minimum network bandwidth of responding servent; search criteria
QueryHit	Returned by servents that have the requested file	IP address, port number, and network bandwidth of responding servent; number of results and result set
Push	File download requests for servents behind firewalls	Servent identifier; index of requested file; IP address and port to send file to

Table 2.1: Message types in the Gnutella protocol

These messages are routed by all servents using a constrained broadcast mechanism: Upon receipt of a message the servent will decrement the message's time-to-live (TTL) field. If the TTL (typically by default set to 7 initially) is greater than 0 and it has not seen the message's identifier before (loop detection) it resends the message to all the peers it knows. Additionally the servent checks whether it should respond to the message. For example, if it receives a *Query* message it checks its local file store and if it can satisfy the request, responds with a *QueryHit* message. Responses are routed along the same path as the originating message.

A simplified Gnutella "session" would work as follows: Servent A connects to servent B and sends a *Ping* message. B responds with a *Pong* and forwards the *Ping* as described above (e.g., it forwards it to its peers C and D who respond with another *Pong*, and so on). After some time A knows a number of servents and vice versa. It routes messages as described above and may initiate queries. When it receives a *QueryHit* for one of its queries it tries to connect directly to the servent specified in the *QueryHit* and runs a simplified HTTP GET [12] interaction to retrieve the file. In case the requested servent is behind a firewall it may send a *Push* message (along the same way as it received the *QueryHit*) to the firewalled servent. The *Push* message specifies were the firewalled servent can contact the requesting servent to run a "passive" GET session. If both servents are behind firewalls then the download is impossible.

2.2.1 Discussion of Gnutella

From a user's perspective Gnutella is a simple yet effective protocol: Hit rates for search queries are reasonably high, it is fault-tolerant towards failures of servents, and adopts well to dynamically changing "peer populations." However, from a networking perspective, this comes at the price of very high network bandwidth consumption: Search requests are "broadcast" over the network and each node receiving a search request scans its local database for possible hits. This is extremely costly in terms of network bandwidth and causes high search costs and long response times.

For example, assuming a TTL of 7 and an average of 4 connections C per peer (i.e., each peer forwards messages to 3 other peers) the total number of messages originating from one Gnutella message (including the responses) can be calculated as $2 * \sum_{i=0}^{TTL} C * (C-1)^i = 26240$. Recent experiments [27] have shown that in a real-world setting this adds up to a network bandwidth consumption of 3.5Mbps (or 353,396 queries in 2.5 hours as another experiment's result given by the same author). To remedy this [27] suggests to apply caching which can reduce the traffic up to 3.7 times.

Also, no estimates on the durations of queries and no probability for successful search requests can be given. Very little is known about the topology of the Gnutella network which would provide the foundation for an accurate mathematical model and would aid the development of new algorithms by allowing them to exploit structural properties and thus be more efficient. As a first step a recent study [21] investigated Gnutella's topology and has shown that it exhibits strong small-world properties [22] and a power law distribution of node degrees.

Besides these technical problems non-technical ones such as "free riding" challenge Gnutella [6]. "Free riding" means that most Gnutella users do not provide files to share and if, only a very limited number is interesting, i.e., being downloaded. [6] shows that nearly 70% of Gnutella users share no files and nearly 50% of all responses are returned by the top 1% of the sharing hosts. This social problem starts to transform Gnutella into a C/S like system which soon may have to face the technological (degradation of performance, vulnerability, etc.) and legal problems of Napster.

Another "social" issue which is not addressed by Gnutella is reputation: In a P2P system peers frequently have to "meet" unknown peers and have no possibility to judge their reputation, i.e., to what extent they can trust the peers and the data provided by them. As shown in [3] Gridella's P-Grid approach may also be used to address this issue efficiently.

These shortcomings of the Gnutella approach—despite its undoubtable merits and the very short development time of the system—motivated the development of the P-Grid-based Gridella system as described in the following sections.

Chapter 3

Conceptual Foundations

3.1 The P-Grid Approach

The underlying idea of the P-Grid approach [1] is to create a virtual binary search structure with replication that is distributed over the peers and supports efficient search, i.e., search time and number of generated messages grow $O(\log n)$ with the number of nodes n in the network. This approach is comparable to other approaches such as [20, 25, 31] to construct scalable, tree-based, distributed indexing structures. However, a main innovation is that the construction and the search/update operations can be performed without any central control and/or global knowledge in an unreliable environment. This is achieved by applying randomized, distributed algorithms. All algorithms for creating and using the search structure are realized through completely decentralized cooperation among the peers. Consequently this search structure exhibits the following properties:

- it is completely decentralized;
- all peers serve as entry points for search;
- interactions are strictly local;
- it uses randomized algorithms for access and search;
- probabilistic estimates for the success of search requests can be given;
- search is robust against failures of nodes; and
- it scales gracefully in the total number of nodes and data items.

3.1.1 The P-Grid Search Structure and Algorithm

P-Grid is a virtual distributed binary search tree which is distributed among a community of peers. This means that each peer only holds part of the overall tree which comes into existence only through the cooperation of the individual peers. The position of every participating peer is determined by its "path," i.e., the binary bit string representing the subset of the overall information in the tree that the peer is responsible for. For example, the path of *peer*4 in the example P-Grid shown in Figure 1 is '10' which means that it is responsible for all data items whose key begins with '10', i.e., stores them.

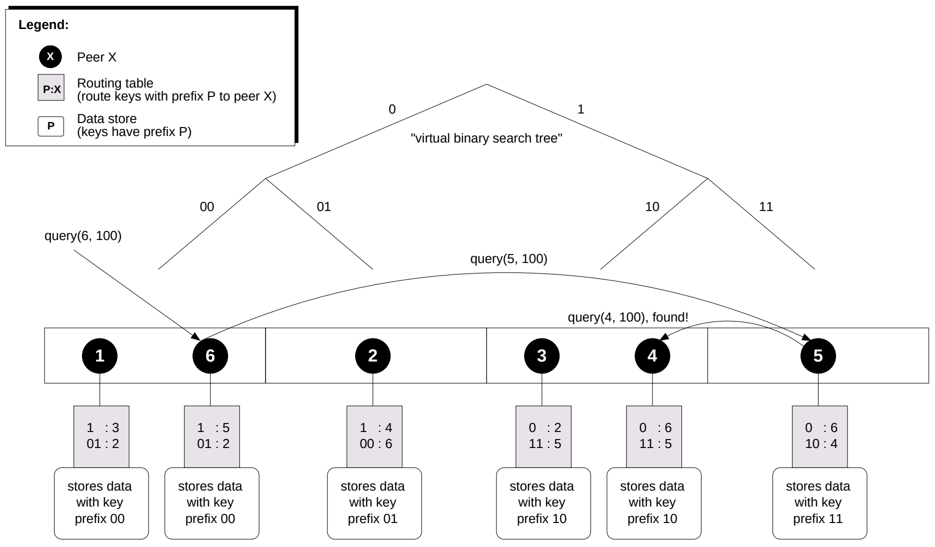

Legend:
X
Peer X
P:X
Routing table
(route keys with prefix P to peer X)
P
Data store
(keys have prefix P)
0
1
"virtual binary search tree"
00
01
10
11
query(6, 100)
query(5, 100)
query(4, 100), found!
1
6
2
3
4
5
1 : 3
01 : 2
1 : 5
01 : 2
1 : 4
00 : 6
0 : 2
11 : 5
0 : 6
11 : 5
0 : 6
10 : 4
stores data with key prefix 00
stores data with key prefix 00
stores data with key prefix 01
stores data with key prefix 10
stores data with key prefix 10
stores data with key prefix 11

Figure 3.1: Example P-Grid

The paths implicitly partition the search space and define the structure of the virtual binary search tree. Its construction will be explained in the next section. As can be seen from Figure 3.1 multiple peers can be responsible for the same path. For example, *peer*1 and *peer*6 are both responsible for keys beginning with '00'. Such replication is up to the individual peers and improves the robustness and responsiveness of the P-Grid since we assume that peers are not online all the time but with a certain, possibly low probability.

As each peer should be able to serve as entry point for any search query, the peers must also store routing information. This means that if a peer is presented with a binary query string and cannot satisfy the query itself, it must forward the query to a peer which is "closer" to the result. P-Grid's routing approach is simple but efficient: For each bit in its path each peer stores the address (Global Unique Identifier + IP address + port number; 1 computer can host multiple peers) of at least one other peer who is responsible for the other side of the binary tree at this level.

For example, *peer*1 will forward queries starting with '1' to *peer*3 which is in *peer*1*'s* routing table and whose path starts with '1'. *Peer*3 may either be able to satisfy the query or forward it to another peer depending on the following bits of the query. If *peer*1 gets a query starting with '0' then it could be responsible since its path also starts with '0'. To decide this, *peer*1 looks at the next bit of the query and if it is '0' it is responsible for the query. Otherwise it will check its routing table and forward the query to *peer*2 whose path starts with '01' which matches the prefix of the query. Figure 3.1 shows the routing tables as grey boxes.

The P-Grid construction algorithm which will be explained in the next section guarantees that the routing tables of the individual peers are constructed in a way that there always exists at least one path between any two peers of a P-Grid which means that any query can be satisfied regardless which peer is queried. This property must be seen "relative" to replication: For example, in Figure 3.2 that visualizes this property for the P-Grid of Figure 3.1, no path between *peer*3 and *peer*1 exists, but there is a path from *peer*3 to *peer*6 which holds the same data as *peer*1.

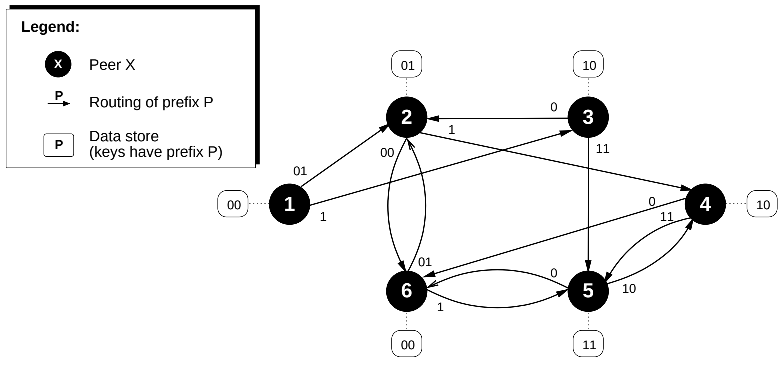


Figure 3.2: Example P-Grid network

A search request in P-Grid works as follows: The query is sent to an arbitrary peer. In Figure 3.1 a query for '100' is sent to *peer*6. Since *peer*6 is responsible for keys starting with '00' it checks its routing table for the longest common prefix with the query which is '1' and forwards the query to *peer*5 that is given in the routing table. In a real setup multiple peers would be listed for each prefix in the routing table and the peer to forward the query to would be chosen randomly. Without constraining general applicability we assume in this simple example that each prefix is "serviced" by one peer entry in the routing table.

Upon receiving the query *peer*5 that is responsible for prefix ‘11’ does the same checks as *peer*6 before and finds out that the query is to be forwarded to *peer*4 which is the longest common prefix in *peer*6*'s* routing table. *Peer*4 in turn has no longer common prefix in its routing table so it is clear that it must search in its local data store for data with the key ‘100’. If the key exists a reference to the associated data is returned to the original requester *peer*6 that may then request the data. This example demonstrates that the order of the search is equivalent to a binary tree search regardless of the entry point of the query.

The algorithm to process a query is shown in Figure 3.3.

```
search (peer, query, index) {
  found = NULL;           /* found: the address of the responsible peer */
  rempath = sub_path(path(peer), index+1, length(path(peer)));
  compath = common_prefix_of(query, rempath);
  IF length(compath)=length(query) OR length(compath)=length(rempath) THEN
    found = peer;
  ELSE
    new_query = sub_path(query, length(compath) + 1, length(query));
    refs = get_refs(index + length(compath) + 1);
    WHILE |refs| > 0 AND NOT found
      ref = random_select(refs);
      IF online(ref)
        found = search(ref, new_query, index + length(compath));
  RETURN found;
}
```

Figure 3.3: P-Grid search algorithm

The parameter `peer` indicates the address of the peer to send the query to, `query` is the search string, and `index` indicates the progress of the search, i.e., how many bits of the query have already been processed. Initially `index` is 0. `sub_path(string, from, to)` returns the substring of `string` that starts at position `from` and ends at position `to`. `common_prefix_of(str1, str2)` returns the common prefix of the two strings `str1` and `str2`. `get_refs(index)` returns the list of addresses in the routing table for a prefix of length `index`. `random_select(refs)` returns an address from this list and removes it from `refs`. `online(ref)` returns true if the referenced peer is online.

The algorithm first compares the common prefix of the peer’s path and the query submitted. As the query string could already be truncated by the first `index` bits, the path of the peer must also be adapted. This is an optimization because at level `index` of the virtual search tree the equality of the first `index` bits is guaranteed. Only the following bits are relevant (line 3) and must be compared with the query to find their common prefix (line 4). If the common path (`compath`) is as long as the query or the remaining path then the peer that is responsible for this query was found (line 5).

Otherwise the query must be forwarded. The common prefix is stripped off the query (line 8), the routing table is queried for the list of peers to forward the query to (line 9) and then the remaining query `new_query` is forwarded recursively to a random peer from this list (if it is online) until the list is exhausted or the search has succeeded (lines 10–13).

3.1.2 Extended Examples

Two simple examples are given in the following to demonstrate the the search algorithm in greater detail. These examples assume the P-Grid given in Figure 3.1.

At first, the query '00' is submitted to *peer*1. `addr1` indicates the address of *peer*1. `Index` is 0 for the initial call.

```
search(addr1, '00', 0) {
```

In the next step the remaining path `rempath` will be computed by the function `sub_path` in line 3. The path of *peer*1 is '00' and therefore the length of the path is 2. `sub_path` will return the bits from 1 (`index + 1`) to 2 of the path, this is '00', the whole path.

```
rempath = sub_path('00', 0 + 1, 2);              // rempath = '00'
```

Then `rempath` will be compared with the submitted query `query` to find a common prefix. The function `common_prefix` in line 4 will return '00'.

```
compath = common_prefix_of('00', '00');          // compath = '00'
```

In line 5 the common path `compath` is compared for equality with the query `query` and the remaining path `rempath`. In this example both pairs are equal, which means that this peer is responsible for the submitted query.

```
IF length('00')=length('00') OR length('00')=length('00') THEN
   found = addr1;
```

At last, the processing peer returns its address to the calling routine in line 14.

```
RETURN found;
```

As a second example, the query '10' is submitted to *peer*6. The lines 1, 3 and 4 will be computed as in the first example.

```
search (addr6, '10', 0) {
   rempath = sub_path('00', 0 + 1, 2);        // rempath = '00'
   compath = common_prefix_of('10', '00');    // compath = ''
```

As we see in this example their will be no equality in line 5. This means, that this peer is not responsible for the query, and it must send it to another peer (else branch).

```
IF length('')=length('10') OR length('')=length('00') THEN
```

First, the new remaining query which is sent to other peers is constructed in line 8, depending on the common prefix of the processing peer. In this example, the new query `new_query` will be the original query, because there is no common prefix.

```
new_query = sub_path('10', 0 + 1, 2);        // new_query = '10'
```

In line 9, references to peers which could be responsible for this query are selected. This is stored in the routing table of the peer (3.1). For this peer, it is only the address of *peer*5 at position 1.

```
refs = get_refs(0 + length('') + 1, addr6);  // refs = {addr5}
```

Because there is at least one reference and the search is not finished successfully yet (line 10), one of the references is selected randomly (line 11). In this case this can only be *addr*5. After selecting it is removed from the list of possible peers to contact.

```
  WHILE |{addr5}| > 1 AND NOT found
     r = random_select({addr5});                     // r = addr5
```

If *peer*5 is online (line 12), the new query `new_query` is sent. The `index` will be still 0, because there is no common path between this peer's path and the submitted query.

```
IF online(addr5)
   found = query(addr5, '10', 0 + length(''));
```

Since there are no more peers to contact, the result of the search algorithm at *peer*5 will be returned to the calling routine at line 14.

```
  RETURN found;
```

The following search routine will be processed at *peer*5.

```
  search(addr5, '10', 0) {
     found = FALSE;
     rempath = sub_path('11', 0 + 1, 2);             // rempath = '11'
     compath = common_prefix_of('10', '11');         // compath = '1'
     IF length('1') = length('10') OR length('1') = length('11') THEN
```

This peer is not responsible for the query either, so it must create a new query and forward it to the available peers stored in the routing table. In this example, the resulting new query is '0' because there is a common prefix with the path of *peer*5 and the original query. So `new_query` is sent to *peer*4 with `index` 1, which means, that we are already at the first level of the search tree.

```
IF length('11') > 0 + length('1') THEN
   new_querypath = sub_path('10', 1 + 1, 2);       // new_query = '0'
   refs = get_refs(0 + length('1') + 1, addr5);    // refs = {addr4}
   WHILE |{addr4}| > 1 AND NOT found
      r = random_select({addr4});                  // r = addr4
     IF online(addr4)
        found = query(addr4, '0', 0 + length('1'));
   RETURN found;
```

As *peer*5 was not responsible for the original query '10' either it replys the answer of *peer*4 to its caller *peer*6. The processed search routine at *peer*4 is shown below.

```
query(addr4, '0', 1) {
   found = FALSE;
   rempath = sub_path('10', 1+1, 2);               // rempath = '0'
   compath = common_prefix_of('0', '0');           // compath = '0'
   IF length('0') = length('0') OR length('0') = length('0') THEN
      found = addr4;
```

Finally *peer*4 is found to be responsible for query '10' and it responds to *peer*5, which calls the routine on *peer*4, with its address *addr*4 to indicate that *peer*4 is responsible for this query. The received address of *peer*4 *addr*4 is forwarded to *peer*6, which is the starting point of the search request. At last *peer*6 returns the address of *peer*4 to the calling routine.

3.2 P-Grid Construction

Having introduced the access structure and the search algorithm an important question remains: How is the P-Grid to be constructed? As there exists no global control this has to be done by using exclusively local interactions. The idea is that whenever two peers meet, they use the opportunity to create a refinement of the access structure. At this point we do not care why and how peers meet. They may meet randomly because they are involved in other operations or because they systematically want to build the access structure. But assuming that by some mechanisms they meet frequently the process works as follows.

Initially, all peers are responsible for the whole search space, i.e., all search keys. At that stage, when two peers meet initially, they decide to split the search space into two parts and take over responsibility for one half each. They also store the reference to the other peer in order to cover the other part of the search space. The same happens whenever two peers meet that are responsible for the same path. While the P-Grid develops also other cases occur: Peers will meet whose paths share a common prefix or whose paths are in a prefix relationship. In the first case the peers can initiate new exchanges by forwarding each other to peers they are referencing themselves. In the second case the peer with the shorter path can specialize by extending its path. To obtain a balanced P-Grid it will specialize in the opposite way the other peer has already done at that level. The other peer remains unchanged.

These considerations give rise to the algorithm in Figure 3.4 that two peers $peer1$ and $peer2$ execute when they meet. `count(data)` returns the number of `data` items. `select(data, key)` returns the `data` with the given `key`. `merge(refs1, refs2)` returns the union of the references `refs1` and `refs2`. If more references are available than the peer wants to manage, a randomly selected fraction is returned.

The algorithm initially constructs the union of the data items managed by the two peers, and counts the data items (line 2 and 3). If a peer's path is empty and the amount of data items is greater than the amount every peer wants to store, the peer extends its path by a random bit (line 4 to 7). Thereby peers only specialize when the amount of data items makes it necessary. The random bit is returned by a function, that guarantees that each peer will extend with inverse bit the other peer would extend. This guarantees the construction of a balanced tree. Now the common prefix of the two paths and its length can be created (line 8 and 9). If the paths are equal and the amount of data with a key of the common prefix is greater than a peer wants to manage, the peers specialize by appending their paths with a random bit (line 10 to 13). Again, the two extension-bits are inverse. If one path is a prefix of the other path, the shorter path must specialize (line 16). Therefore a string of random bits is generated with the length of the difference between the two paths (line 16 and 24). This string is now compared with the longer path ignoring the leading common prefix, and the common prefix of these two strings is appended to the shorter path (line 19, 20 and 25, 26). If the shorter path is still shorter than the longer path, it is further extended by the next bit of the generated string of random bits, which is not equal to that of the longer path (line 21, 22 and 27, 28).
For example, let $path1$ be '01001010011' and $path2$ be '01001', then '01001' is the common prefix and $path2$ is a prefix of $path1$. The generated string of random bits (ext) could be '010110'. Now this string must be compared with the substring '010011', the rest of $path1$ after the common prefix. The common prefix ($comExt$) for these two string is '010', which will be appended to $path2$. As $path2$ is still shorter than $path1$, the next bit of the generated string of random bits (ext) is

```
exchange(peer1, peer2) {
  data = is the union of the data items managed by the two peers
  items = count(data)
  IF path(peer1) is empty AND items > the peer wants to manage
    extend path(peer1) with a random bit
  IF path(peer2) is empty AND items > the peer wants to manage
    extend path(peer2) with the inverse random bit
  compath = common prefix of the two paths
  len = the length of the compath
  IF path(peer1) = path(peer2)            /* paths are equal */
    IF count(select(data, compath)) > the peer wants to manage
      extend path(peer1) with a bit
      extend path(peer2) with the inverse random bit
  len1 = length(path(peer1)) - len
  len2 = length(path(peer2)) - len
  IF len1 = 0 OR len2 = 0                 /* one path is a prefix of the other */
    IF len1 < len2
      ext = a string of random bits of length len2
      comExt = the common prefix of ext and the rest of path(peer2) after compath
      extend path(peer1) with comExt
      IF ext != comExt
        extend path(peer1) with the next bit of ext after compath
    IF len2 < len1
      ext = a string of the inverse random bits of length len1
      comExt = the common prefix of ext and the rest of path(peer1) after compath
      extend path(peer2) with comExt
      IF ext != comExt
        extend path(peer2) with the next bit of ext after compath
  len = the length of the common prefix of the (changed) two paths
  IF len > 0                              /* common prefix exists */
    FOR i = 0 TO len
      refs(peer1)[i] = merge(refs(peer1)[i], refs(peer2)[i])
      refs(peer2)[i] = merge(refs(peer2)[i], refs(peer1)[i])
  IF path(peer1) = path(peer2)            /* paths are equal */
    replicas(peer1) = merge(replicas(peer1), {peer2, replicas(peer2)})
    replicas(peer2) = merge(replicas(peer2), {peer1, replicas(peer1)})
  ELSE
    refs(peer1)[len] = merge(refs(peer1)[len], {peer2, replicas(peer2)})
    refs(peer2)[len] = merge(refs(peer2)[len], {peer1, replicas(peer1)})
  data(peer1) = select(data, path(peer1))
  data(peer2) = select(data, path(peer2))
  try to exchange to all new hosts found in the references and replicas
}
```

Figure 3.4: P-Grid construction algorithm

also appended, in this case ‘1’. So *path*2 is now ‘010010101’.
After the paths have been changed, the new length of the common path of the two paths is calculated (line 29). If a common prefix exists, the references of the two peers are merged at all levels of the common prefix (line 30 to 33). If the two paths are equal, the peers reference each other in their list of replicas. Additionally the replicas of the other peer are added. (line 34 to 36). If the paths are not equal, the peers reference each other and the replicas of the other peer at the level the first bit of the paths differ (line 37 to 39). At last each peer selects the data items with keys matching its path from the initially constructed union of data items (line 40 and 41).
The peers use the list of references and replicas of the other peer to initiate further exchanges with previously unknown peers (line 42).

The path of peers joining the P-Grid structure the first time is empty and is only extended when the amount of data items of both peers makes it necessary, i.e., two peers with an empty path and no or too few data items keep their empty paths. If the amount of data items of both peers is sufficient, then the paths are extended by a random bit to guarantee a balanced P-Grid tree. If the P-Grid tree is extremely unbalanced, the number of messages used to process a query still scales gracefully. [2] demonstrates that the use of balanced tree structures for data indexing is not necessarily required in a P2P environment.

3.3 Mapping File Names into Binary Keys

In the P-Grid approach we assume that search keys have a binary representation and are uniformly distributed. Both assumption, however, do not hold for real filenames. Thus we provide a mapping scheme that calculates a binary representation from a filename string. To support search on these binary keys this mapping has to satisfy a prefix property for strings s_1 and s_2:

$$s_1 \ prefix \ s_2 \Rightarrow key(s_1) \ prefix \ key(s_2).$$

The construction algorithm is given in Figure 3.5.

The algorithm proceeds by first constructing a balanced trie structure based on a sample database of search strings. The trie structure is then used to compute the binary keys for search keys. If the sample database is big enough and represents the distribution of all search strings well and since the trie is balanced, the

```
MakeTrie(sampledb) {
  /* sort the sampledb in lexicographical order */
  SortLex(sampledb)
  /* find the common prefix for all the strings from sampledb */
  commonprefix = CommonPrefix(sampledb)
  /* We choose the middle string from sampledb and take the prefix of
     length commonprefix+1, which is enough to split the sampledb into two
     approximately equal parts. It is possible to explore different
     alternatives at this point, in order to achieve a more balanced split */
  IF Size(sampledb) > MaxLeafStore
    mid = Prefix(sampledb[Quotient(Size(sampledb), 2)], Length(commonprefix) + 1)
  FOR j = 1 TO Size(sampledb)
    IF sampledb[j] is lexicographically smaller than mid
      lowpart = Append(lowpart,sampledb[j])
    IF sampledb[j] is lexicographically greater than mid
      highpart = Append(highpart,sampledb[j])
  IF Size(lowpart) > MaxLeafStore
    left = MakeTrie(lowpart)
  ELSE
    left = null
  IF Size(highpart) > MaxLeafStore
    right = MakeTrie(highpart)
  ELSE
    right = null
  root = mid
  TrieSet(root, left, right)
}
```

Figure 3.5: Trie construction algorithm

```
FindKey (trie, filename) {
  key = {}
  IF trie = null OF filename is prefix or equal to trie.root
    return key
  ELSE
    IF filename is lexicographically smaller than trie.root
      key = Append(key, 0)
      key = Append(key, FindKey(trie.left, filename)
    ELSE
      key = Append(key, 1)
      key = Append(key, FindKey(trie.right, filename)
}
```

Figure 3.6: Mapping strings into binary keys

resulting distribution of encoded binary search strings should be approximately uniform.

Function `MakeTrie` is used for building the trie structure and has one parameter `sampledb` which is the sample search string database. The sample database `sampledb` contains unique strings of length `len` which are substrings of actual search strings. As a result `MakeTrie` returns the trie structure. The trie structure is built in the following way: First `sampledb` is sorted in lexicographical order and then is split into two approximately equally sized parts. The split is performed by taking the shortest possible string such that the lower part contains the strings that are lexicographically smaller than the string and the higher part contains the strings that are lexicographically greater than the string. Then the value of this shortest string is stored in the root of the tree and the function is called recursively for both resulting parts of `sampledb` where the lower part corresponds to the left branch and the higher part corresponds to the right branch of the tree. The splitting proceeds until there are less or equal than `MaxLeafStore` strings in the database.

Once the trie has been constructed, it can be used for mapping strings (filenames) into binary keys as shown in Figure 3.6.

Function `FindKey` has two parameters: the trie structure `trie` and the string `filename`. It returns the binary key that corresponds to `filename`. The binary key is calculated in the following way: The string is lexicographically compared to the root value of the trie and if it is prefix of or equal to the root then the calculation terminates returning the binary key; otherwise if it is smaller, then '0' is appended to the key and the function is called recursively with the left subtree; if it is greater '1' is appended to the key and `FindKey` is called with the right subtree.

3.3.1 Experiments

We implemented the mapping algorithm in Java and modified the open source Gnutella client Furi [30] to monitor and log all queries that were routed through it. We used this data to construct a large database of Gnutella queries to evaluate the quality of the mapping algorithm. From the query database we derived a sample database that we used for constructing the trie structure. Then we tested the trie structure by encoding the complete set of search strings and verifying that resulting keys were uniformly distributed. We give one exemplary result to illustrate this: Of 33799 search strings of length 4 ($len = 4$) which were logged with Furi we randomly selected 1951 strings for the sample database. We used this set to construct the trie using $MaxLeafStore = 30$ which resulted in 99 different keys. When generating the keys for all search strings, the maximum number of strings mapped to one key is 798. A perfectly uniform distribution would result in a maximum of 342 search strings per key such that in the worst case slightly more than twice the number of search strings are encoded into the same key as with a perfectly uniform encoding. Thus the resulting distribution is of fairly good quality with respect to uniformity and the workload for peers for storing data and answering queries will be distributed approximately uniformly as well. This is also illustrated by the frequency histogram shown in Figure 3.7. It shows that we achieve almost a normal distribution.

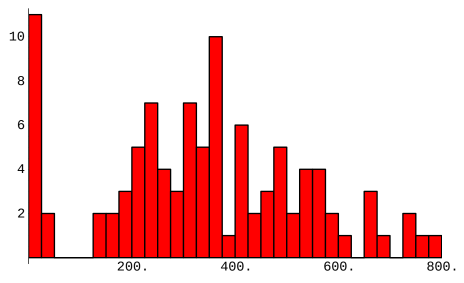


Figure 3.7: Key frequencies in steps of 25

Chapter 4

Architecture

Gridella is our Gnutella compatible P2P system which is based on the P-Grid approach described in the previous section. It is intended to gradually evolve the existing Gnutella infrastructure into a technologically more advanced system. The goals of the Gridella system are:

- compatibility with Gnutella
- a smooth migration path to a fully Gridella based infrastructure
- a more efficient distributed data storage and search strategy
- reduction of network traffic
- a modular design which supports reuse and extensibility
- support for multiple communication protocols

Several proof-of-concept implementations and simulations were done before the current Gridella implementation written in Java. It will be released to the public domain under the GNU General Public License and can be installed and used easily by users of other Gnutella clients. Developers wanting to use the current implementation (e.g., to develop some nice GUIs) can get it upon request.

4.1 Component Model

Figure 4.1 shows the main components of the Gridella system.

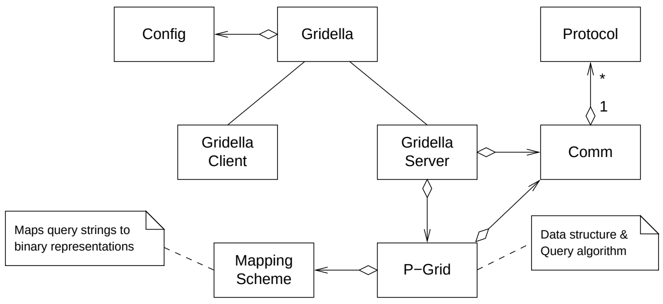


Figure 4.1: Gridella core system components

The components fall into two categories: The Gridella client which provides all user-related functionality and the Gridella server that handles data management and communication. A GUI is not included in Figure 4.1 because I want to provide a high-level library that abstracts from and handles all Gridella related functionality (the implementation includes a GUI, of course). Such I have a simple 2-layer architecture that enables multiple GUI implementations. Inside the core system I follow this rationale, too, so that only the client or the server could be reused by other software.

The main task of the Gridella client is to interface the user with the server and provide special services to the user (e.g., to play the shared and downloaded media files). The server part implements the P-Grid algorithm and communicates with other peers via the communication subsystem which provides communication abstractions for any number of on-the-wire protocols. At the moment I support the Gnutella protocol and the internal Gridella protocol. By separating communication from the protocols I offer a simple way to include new protocols without having to change the overall system. Thus Gridella may use existing protocols, for example, of other P2P systems or provide tunneling via HTTP, or benefit from new, possibly more efficient protocols. As a side effect this enables Gridella to communicate with multiple different systems at the same time which means that it also offers some kind of gateway functionality. The Mapping Scheme subsystem of P-Grid maps all user search requests into requests the search algorithm of P-Grid can process.

4.2 Communication Model

4.2.1 Searching

The communication model of Gridella is shown in Figure 4.2 for the special case of a search interaction between two Gridella peers.

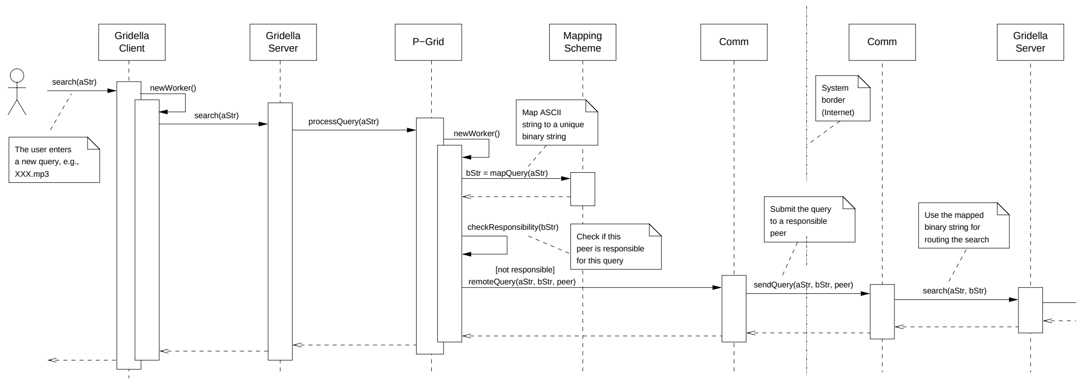

Gridella Client
Gridella Server
P−Grid
Mapping Scheme
Comm
Comm
Gridella Server
search(aStr)
newWorker()
The user enters a new query, e.g., XXX.mp3
search(aStr)
processQuery(aStr)
newWorker()
Map ASCII string to a unique binary string
bStr = mapQuery(aStr)
checkResponsibility(bStr)
Check if this peer is responsible for this query
[not responsible]
remoteQuery(aStr, bStr, peer)
System border (Internet)
Submit the query to a responsible peer
sendQuery(aStr, bStr, peer)
Use the mapped binary string for routing the search
search(aStr, bStr)

Figure 4.2: A Gridella Search interaction

The user initiates the query by providing a query via the GUI. The GUI calls the according method of the Gridella client which creates a new worker thread, forwards the query to the worker, and again waits for further requests. The worker in turn forwards the request to the Gridella server component, which initiates the actual search process by requesting a search operation from the P-Grid component.

At first the P-Grid component creates a new worker to handle the request. The worker in turn requests the mapping described in Section 3.3 to convert the original query into a binary representation feasible for P-Grid. Then it checks if it is responsible for the query, i.e., whether it has the requested information available locally. If so, it returns it. If not, it determines which peer to contact according to the P-Grid algorithm as described in Section 3.1 and contacts this peer via the communication subsystem. The communication subsystem contacts its counterpart at the requested peer using the defined protocol, e.g., via the P-Grid protocol, and sends the query. At the recipient the query is forwarded to the Gridella server and the same interaction occurs again.

Interacting with a non-Gridella peer, for example a Gnutella peer, is also simple: In this case the Gridella peer would forward the original query (not mapped) to the Gnutella peer via the Gnutella protocol. In terms of Figure 4.2 this means that the Gridella server would not request a mapping, check local availability of the requested information and possibly instruct the communication subsystem to contact the Gnutella peer via the Gnutella protocol. This would be the standard interaction pattern of a Gnutella peer employing Gnutella's undirected search approach. To distinguish Gridella-enabled peers which can employ the advanced capabilities of Gridella the system keeps lists of its communication partners and their capabilities.

As stated above, if the search was successful the requested information is returned. This requires a little bit of further consideration since it depends on the employed protocols. For example, in Gnutella a reference would be returned first and then the information download would be done via a HTTP-like GET request or a PUSH operation as described in Section 2.2. This works if at least one of the peers is not behind a firewall. Other protocols might simply return a URL where the information could be downloaded or transfer the information via the connection that already had been established by the remote query request (this would solve the firewall problem of Gnutella). We plan to include a flexible mechanism so that Gridella can benefit from the advantages of specific protocols.

4.2.2 Meetings

The meeting of two peers leads to the second interesting interaction, shown in Figure 4.3.

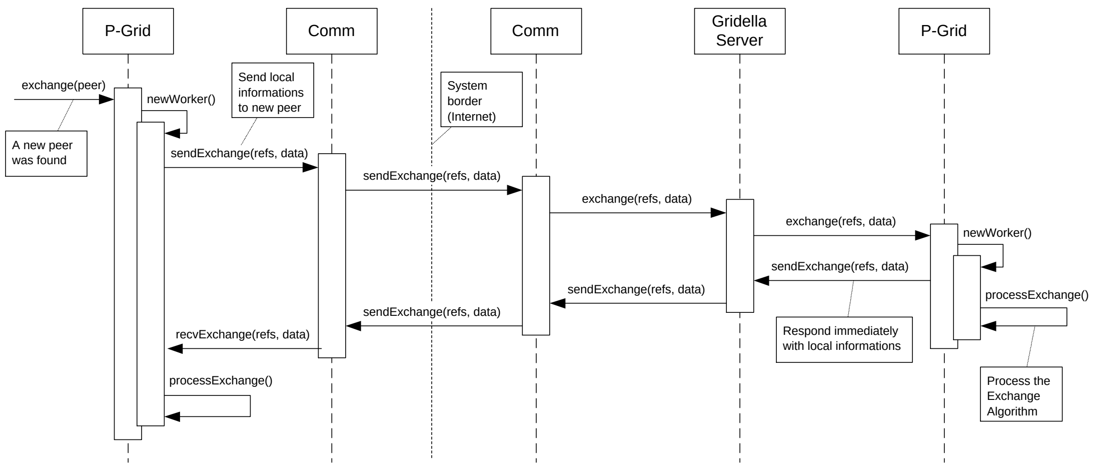


Figure 4.3: A Gridella Exchange interaction

Whenever a new peer is found an Exchange is done to evolve the P-Grid infrastructure. In P-Grid peers must exchange periodically after the first meeting to continuously evolve and maintain the P-Grid structure. Thereby the references of peers are kept up-to-date and new data items spread out over the peers and become accessible for search requests. Peers joining the P-Grid structure (the first time, or each application startup) extend the exchanged list of locally managed files (a part of all files shared in the P-Grid infrastructure) with the local shared files. Thereby new peers initially register their files and possible "lost" files are registered again. A file can get unavailable, when the only managing peer becomes offline. The period between two Exchanges of two peers depends on the number of already successfully processed Exchanges. "Newer" peers with shorter paths exchange more frequently than "older" hosts with very specialized paths. First an Exchange message containing the path, all references and all managed data items of the local peer is sent over the communication subsystem. Its counterpart at the requested peer will forward the received Exchange request to the P-Grid component and responds immediately with its local P-Grid information (path, references, and data items). To speed up the Exchange interaction, the response is sent immediately by the requested peer before it starts the Exchange algorithm (described in Section 3.2). Thereby the requesting peer receives the response sooner and can start its Exchange algorithm sooner to shorten execution time. This allows the peers to process more exchanges in a shorter time and to be available for incoming exchange requests (before the request raises a timeout at the requesting host).

4.2.3 Downloading

A download can be requested for all files found by a previous search request of a user. The interactions to perform a download are shown in Figure 4.4.

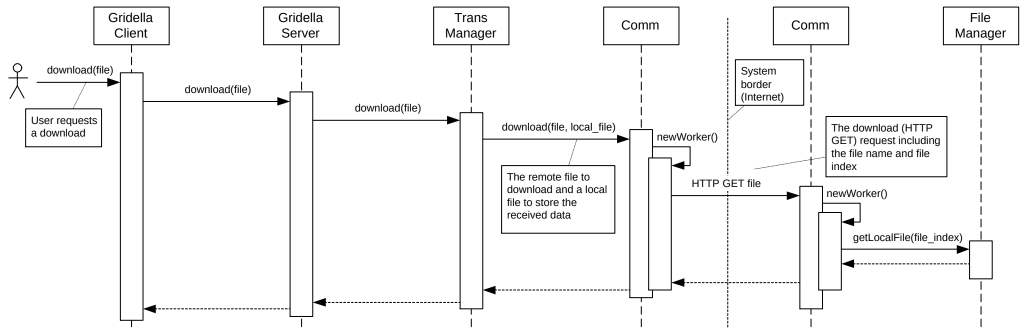


Figure 4.4: A Gridella Download interaction

A user's download requests are forwarded from the Gridella Client to the Gridella Server, and then to the Transmission Manager. The manager creates a temporary file to store the received data of the download and calls the download method of the Communication component with the file to download and the local temporary file. The Communication component starts a new worker for the download and contacts peer storing the file. A HTTP GET message containing the filename and file index indicates the file at the contacted peer. The Communication component of the storing peer uses the File Manager to get the requested file by the file index. Then the Communication components can process the actual download by transferring the file data. The receiving component stores the data into the temporary file and returns this file to the Transmission Manager upon completion. The user is being informed about the download status via the GUI during the whole process.

Chapter 5

Protocol Design

5.1 Requirements

Gridella is designed for interoperability with Gnutella and the long-term goal is to replace the existing Gnutella infrastructure with Gridella. So Gridella implements the existing Gnutella protocol [9] as well as the new Gridella protocol. To achieve full compatibility, the Gnutella protocol was not extended and Gridella could also be used as stand-alone Gnutella servent. Since the software can forward queries between the two protocols Gridella is also a gateway between these two infrastructures.

5.2 The Gridella Protocol

A Gridella servent joins the P-Grid infrastructure by establishing a connection with another servent currently on the network. If the servent joins the network the first time it contacts the servent `www.p-grid.org:1805` to receive addresses of other servents. At the next startups the peer uses the addresses stored in the routing table of the last session. After the connection is established, the following Gridella connection request string is sent:

```
GRIDELLA CONNECT/<protocol version>\n\n
```

where <protocol version> is currently defined as "`1.0`".

A servent wishing to accept the connection request must respond with:

```
GRIDELLA OK\n\n
```

Any other response indicates the servent's unwillingness to accept the connection. If the connection is refused, Gridella tries to connect via the Gnutella protocol (see the Gnutella Protocol [9] specification for details) to establish a connection to the Gnutella network. If this fails too the servent is no Gridella or Gnutella servent or currently not accepting incoming connections.
Once a servent has connected successfully, it communicates with the other servents by sending and receiving Gridella protocol messages. All Gridella messages are XML-based. There DTDs are given in the following.
All Gridella messages must start with a `<Gridella>` tag which must contain at least a `<Host>` element. The message type is given by the next element. This could be `<Exchange>`, `<FileRegister>`, `<Query>`, `<QueryReply>` or `<Push>`. Figure 5.1 shows the DTD for a minimal Gridella message.

```
<!DOCTYPE Gridella [
        <!ELEMENT Gridella (Host, (Exchange|FileRegister|Query|QueryReply|Push)*)>
        <!ATTLIST Gridella
                  Version  STRING  #REQUIRED>
        <!ELEMENT Host     EMPTY>
        <!ATTLIST Host
                  IP       STRING  #REQUIRED
                  Port     INTEGER #REQUIRED>
]>
```

Figure 5.1: The Gridella message DTD

The attributes of `<Gridella>` are:

- `Version` is the version of the protocol version, currently "1.0".

The attributes of `<Host>` are:

- `IP` is the sending host's IP address.
- `Port` is the port for Gridella messages of the sending host.

5.2.1 Exchange

An Exchange message is used by peers of the P-Grid network to establish and evolve the network. That means that every peer gets its path and its data items to manage. When two peers meet the first time an Exchange message is sent to the other peer, and then repeatedly after a certain interval. A peer receiving an Exchange message must response with an Exchange message. The message contains the path, all stored references to other peers of the sending host and the managed data items. The path is given bitwise for each level (`Index`) of the P-Grid structure (see Section 3.1.1), and for each level the corresponding references are given. There could also be references to replicas included.

```
<!DOCTYPE Exchange [
        <!ELEMENT Exchange  (Path*, Replica?)>
        <!ELEMENT Path      (Peer*)>
        <!ATTLIST Path
                  Index     INTEGER #REQUIRED
                  Value     {0,1}   #REQUIRED>
        <!ELEMENT Replica   (Peer*)>
        <!ELEMENT Peer      EMPTY>
        <!ATTLIST Peer
                  IP        STRING  #REQUIRED
                  Port      INTEGER #REQUIRED>
        <!ELEMENT Data      (File*)>
        <!ELEMENT File      EMPTY>
        <!ATTLIST File      STRING  #REQUIRED
                  IP        STRING  #REQUIRED
                  Port      INTEGER #REQUIRED
                  Key       STRING  #REQUIRED
                  Index     INTEGER #REQUIRED
                  Size      INTEGER #REQUIRED
                  Info      STRING>
]>
```

Figure 5.2: The Exchange message DTD

The Exchange message contains a `<Path>` element for each bit of the sending host's path. For example, a message from a host with a path '011' contains the elements `<Path Index=0 Value=0>`, `<Path Index=1 Value=1>`, and `<Path Index=2 Value=1>`. If the host has replicas the message further contains the `<Replica>` element. Each `<Path>` and `<Replica>` element contains one or more references to other hosts represented by the `<Peer>` element. If the sending host manages data items of the P-Grid item, the message contains the `<Data>` element and a `<File>` element for each managed file.

The attributes of `<Path>` are:

- `Index` is the level of the P-Grid structure represented by this element. The first index is ‘0’.
- `Value` is the bit at index `Index` of the sending host’s path. Index ‘0’ represents the most significant bit of the path.

The attributes of `<Peer>` are:

- `IP` is the IP address of the referenced peer.
- `Port` is the port at which Gridella listens of the referenced peer.

The attributes of `<File>` are:

- The file name.
- `IP` is the IP address of the peer storing this file.
- `Port` is the port at which Gridella listens at the peer storing this file.
- `Key` is the key (the binary representation) of the file name.
- `Index` is the index of this file at the storing peer, unique at this host, used in a download request to indicate the file.
- `Size` is the size of this file in bytes.
- `Info` may hold additional informations about this file (e.g. bit rate, resolution, track length in seconds, ...)

5.2.2 Query

A Query message defines a search request. Query messages are only sent to peers, which could be responsible for this query (see the P-Grid search algorithm in Figure 3.3). Peers receiving a query message use the `Key` attribute to decide, if they are responsible for this query or not. The `Key` represents the binary representation of the search string used to route the request to the responsible peer. The `Key` is truncated of the first `Index` bits, when the request was already processed by another peer. Thereby the receiving peer can not verify if its path is a prefix of the original `Key` and therefore verify if it is responsible for this query. If it is responsible it replys with a QueryReply message, otherwise it is forwarding the Query message to a maybe responsible peer.

```
<!DOCTYPE Query [
        <!ELEMENT Query    EMPTY>
        <!ATTLIST Query    STRING  #REQUIRED
                  Index    INTEGER #REQUIRED
                  Key      STRING  #REQUIRED
                  MinSpeed INTEGER>
]>
```

Figure 5.3: The Query message DTD

The attributes of `<Query>` are:

- The query string (e.g., “Madonna”, “The Who mp3”, ...).
- `Index` defines the search progress, the number of truncated leading bits of the `Key`. The search starts with index ‘0’, which means that the `Key` represents the original binary representation of the query string, and is increased with every processed bit of the `Key` derived from the query.
- `Key` is the key (the binary representation) of the query string, or a substring of the key if `Index > 0`.
- `MinSpeed` indicates the minimum speed (in kB/second) responding peers should be able to communicate with.

5.2.3 QueryReply

A QueryReply message is sent in response to a received Query message, if a peer is responsible for the Query, i.e., can answer it. QueryReply messages can hold three different results (`Code` element):

- `200` is returned if files where found for the received query string. Then the QueryReply message must also include a result set of found files.

- `400` is returned if a erroneous Query message was received. The QueryReply element is empty.
- `404` is returned if no files where found for this query string. The QueryReply element is empty.

```
<!DOCTYPE QueryReply [
        <!ELEMENT QueryReply (Result*)>
        <!ATTLIST QueryReply
                  Code       INTEGER #REQUIRED>
        <!ELEMENT Result     EMPTY>
        <!ATTLIST Result     STRING  #REQUIRED
                  IP         STING   #REQUIRED
                  Port       INTEGER #REQUIRED
                  Index      INTEGER #REQUIRED
                  Size       INTEGER #REQUIRED
                  Info       STRING>
]>
```

Figure 5.4: The QueryReply message DTD

The attributes of `<QueryReply>` are:

- `Code` indicates the type of the message (200, 400, or 404).

The attributes of `<Result>` are:

- The file name.
- `IP` is the IP address of the peer storing this file.
- `Port` is the port at which Gridella listens at the peer storing this file.
- `Key` is the key (the binary representation) of the file name.
- `Index` is the index of this file at the storing peer, unique at this host, used in a download request to indicate the file.
- `Size` is the size of this file in bytes.
- `Info` may hold additional informations about this file (e.g. bit rate, resolution, track length in seconds, ...)

5.3 The Protocol Automaton

The protocol automaton in Figure 5.5 shows the sequence of messages sent and received by a peer. Depending on how the connection was created, by the local host (outgoing) or by another host (incoming), the connection is connected or accepted. Connecting connections send the Init message and wait for a response. Accepting connections receive the Init message, reply with the Init response message and wait for further messages. As soon as the connecting host receives the Init response message, it also waits for further messages. In this state, only an Exchange message can be sent or received, or a Query message is sent or received. When a message is received by the other host, the corresponding reply is sent - an Exchange message in reply to an Exchange message, and a QueryReply message in reply to an Query message. If a message is sent from the local host the corresponding reply message is received - an Exchange message in reply to an Exchange message, and a QueryReply message in reply to an Query message. After any of these message interactions it waits for further messages.

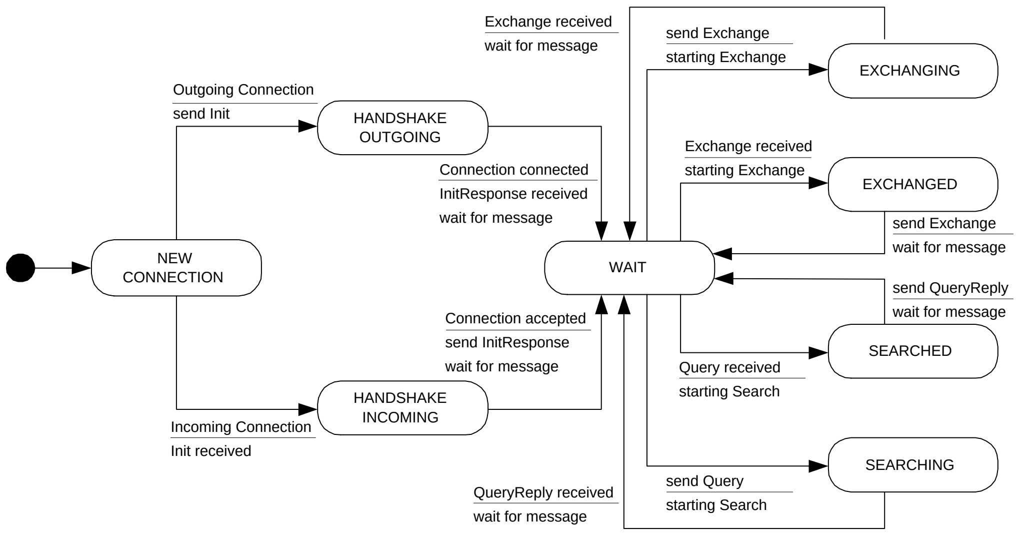


Figure 5.5: The protocol automaton

5.4 A typical Protocol Interaction

This section shows two typical Protocol Interactions to demonstrate how the Gridella protocol works. The first example shows what happens, when two peers meet the first time. The second example shows a simple search request and its response.

5.4.1 First Meeting

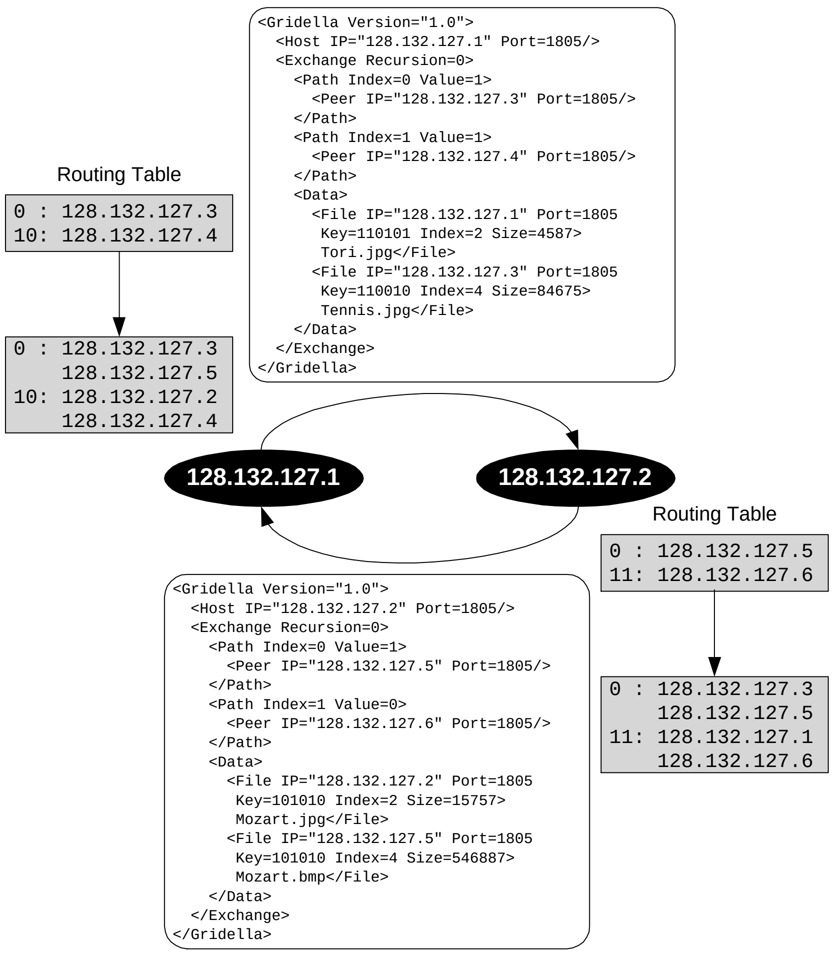
When two peers meet for the first time, each of the two peers sends an Exchange message. Thereby each peer gets to know the other peer's path, its references and managed data items. This is necessary for all peers to construct the Gridella grid and to be able to satisfy all user search requests. A peer's path is accompanied by a list of peers known to this peer. This enables all peers to detect new peers.

In Figure 5.6 two peers, `peer1` with IP address `128.132.127.1` and `peer2` with IP address `128.132.127.2` both listening at port `1805` meet the first time and send Exchange messages. The path of `peer1` is '11' and `peer2`'s is '10'. Therefore both messages contains two `<Path>` elements, each representing one bit of the path, and the corresponding references. Each of them knows two further peers, as mirrored by their routing tables. `Peer1` manages the files *Tori.jpg* and *Tennis.jpg*, `peer2` manages *Mozart.jpg* and *Mozart.bmp*, which are represented by a `File` element in the `Data` element of the messages. As the two paths are not equal or are a prefix of each other, the Exchange algorithm will only lead to an extended routing table, as shown by the changed routing tables. So each peer gets to know two further peers and can contact other peers.

When two peers with the same path meet, the total number of data items managed by these two peers together determines if the two paths are being specialized by appending a random bit. The bits are inverse to each other to construct a balanced grid. Then the data items are divided among the peers according to the key of the files. Each peer manages only files, if the peer's path is a prefix of the key. If the number of data items is to small to specialize, the peers become replicas by merging the references and data items.
Each peer receiving an Exchange message uses all references of the message to discover new peers and exchange with them. Further the data items of the sending host are searched for files the receiving peer's path is a prefix of the file key.

5.4.2 Search Request

Every time a user enters a search request, this will result in Query and QueryReply messages. When a peer is not able to fulfill a request locally, it sends a Query message to a peer which could be responsible for this query.

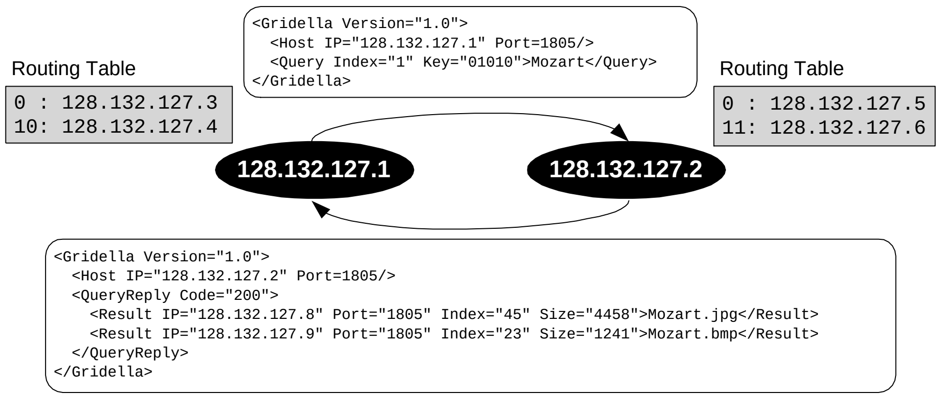


```
<Gridella Version="1.0">
  <Host IP="128.132.127.1" Port=1805/>
  <Exchange Recursion=0>
    <Path Index=0 Value=1>
      <Peer IP="128.132.127.3" Port=1805/>
    </Path>
    <Path Index=1 Value=1>
      <Peer IP="128.132.127.4" Port=1805/>
    </Path>
    <Data>
      <File IP="128.132.127.1" Port=1805
       Key=110101 Index=2 Size=4587>
       Tori.jpg</File>
      <File IP="128.132.127.3" Port=1805
       Key=110010 Index=4 Size=84675>
       Tennis.jpg</File>
    </Data>
  </Exchange>
</Gridella>
```

```
<Gridella Version="1.0">
  <Host IP="128.132.127.2" Port=1805/>
  <Exchange Recursion=0>
    <Path Index=0 Value=1>
      <Peer IP="128.132.127.5" Port=1805/>
    </Path>
    <Path Index=1 Value=0>
      <Peer IP="128.132.127.6" Port=1805/>
    </Path>
    <Data>
      <File IP="128.132.127.2" Port=1805
       Key=101010 Index=2 Size=15757>
       Mozart.jpg</File>
      <File IP="128.132.127.5" Port=1805
       Key=101010 Index=4 Size=546887>
       Mozart.bmp</File>
    </Data>
  </Exchange>
</Gridella>
```

Figure 5.6: Example Exchange Interaction

Figure 5.7 shows two peers with their routing tables. As `peer1` is not responsible for the search request ‘Mozart’ (key ‘101010’), it sends a Query message to `peer2` and waits for a QueryReply message. After `peer2` receives the request from `peer1` it tries to fulfill the query locally. As `peer2` is responsible for this search request, it responds with a QueryReply message including all found files matching the query string.



```
0 : 128.132.127.3
10: 128.132.127.4
```

```
0 : 128.132.127.5
11: 128.132.127.6
```

```
<Gridella Version="1.0">
  <Host IP="128.132.127.1" Port=1805/>
  <Query Index="1" Key="01010">Mozart</Query>
</Gridella>
```

```
<Gridella Version="1.0">
  <Host IP="128.132.127.2" Port=1805/>
  <QueryReply Code="200">
    <Result IP="128.132.127.8" Port="1805" Index="45" Size="4458">Mozart.jpg</Result>
    <Result IP="128.132.127.9" Port="1805" Index="23" Size="1241">Mozart.bmp</Result>
  </QueryReply>
</Gridella>
```

Figure 5.7: Example Search Request Interaction

Chapter 6

Component Design

The architecture of Gridella introduced in Chapter 4 consists of four main components - Client, Server, P-Grid, and Communication - which were designed as self-contained units to support reusability. The Communication component includes two subclasses for each supported protocol (Gnutella and Gridella). The given hierarchical dependencies of the components influenced also the interactions between the components. This allows the replacements of a component without affecting other components. This chapter presents the class diagrams of the component's classes, which can be found in the packages.

6.1 Client Package

The Client package provides the possibility to interact with the Server package. Any user requests received by a user interface (e.g., the included GUI) are tunneled by the Client to the Server package. Thus the `Client` component in the Client package provides lot of services to connect/disconnect other peers, add/remove searches, add/remove downloads and to monitor the Server itself.

The Client package also contains all components used by a user interface. Gridella contains a `Player` component, which allows the user to open a shared or recently downloaded file with certain applications, e.g., Winamp, RealPlayer, Windows Media Player, or Acrobat Reader. The player application must have been parameterized with a file-extension for a certain type to support this functionality.

6.2 General Package

The General package contains data structures for communication between certain components and packages. Further it provides global constants, other configuration parameters and useful utilities.

The **Configuration** (`Cfg`) component provides all global constants and all configuration parameters used by nearly all components. Global constants are the standard listening port for Gridella, the used Gridella protocol version, the rate to perform exchanges with other hosts, or the number of data items required to specialize the P-Grid path. The listening port can be configured by a command line argument. All other configuration parameters are read from the configuration file (`gridella.properties`) and can be accessed by the Configuration component.

The **Connection** (`Connection`) component represents a connection to another host over the network. The connection can be a Gridella, Gnutella, or HTTP connection with a certain status (connecting, accepting, connected, not connected, error). Each connection belongs to a Host component. A Connection is created by the Communication Manager before connecting or accepting a host, and provides informations about the start time of the connection, the sent/received messages and bytes. Each Connection provides a send and receive queue, used by the Communication package to communicate between the different components. For example, the Communication Reader reads messages from the network and put them into the queue. This messages where then taken and processed by the Communication Workers. They transform the received messages into intern data structures and call the corresponding components for further processing. To send a message to another host, a component puts the message into the send queue, which will be taken and written by the Communication Writer.

The **Global Unique Identifier** (`GUID`) component is used to identify objects globally unique, i.e., hosts, files, and messages.

The **Host** (`Host`) component contains general informations, e.g., the host address, the type (Gridella or Gnutella), the path (only for Gridella hosts), the number of shared files, shared kBytes, the connection speed or if the host is behind a firewall. The component is used by P-Grid to represent a reference to peers, by files to determine the storing host, or by the Host Manager to administrate Gnutella hosts.

The package further contains data structures only used intern to represent different messages (Exchange, Query, Query Hit), shared and managed data items (files), and downloads/uploads.

6.3 Server Package

The Server package provides the possibility for other packages to interact with the Server and its defined services using the `Server` component. The requested services are then processed by one of the subclasses of the Server. The architecture of the Server package is given in Figure 6.1.

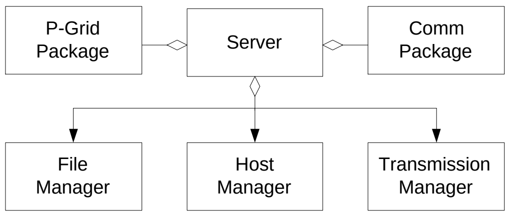


Figure 6.1: The Server architecture

The **File Manager** component (`FileManager`) manages all local shared files and is responsible for all incoming Gnutella search requests. It searches for files matching the query string and informs the Message Manager if matches are found. All files are identified uniquely by the file index given by the File Manager. The index is used in common with the file name by other hosts to request file downloads.

The **Host Manager** component (`HostManager`) manages all Gnutella and Gridella hosts. It administrates lists of both types, which can be used by a GUI to display the network status of the local servent. To be informed about changes of host properties, components can register as “host listeners”. Every time a host property changes, all registered listeners are informed. Gnutella hosts can be contacted manually, e.g., a user enters a host address, or automatically. The user only gives the number of requested established Gnutella connections. Erroneous hosts can also be removed automatically.

The **Transmission Manager** component (`TransManager`) manages all downloads and uploads to/from the local host. Transmissions can be added, removed, aborted, or cleared. Informations about transfers, for example: the average transmission speed or if the manager is busy, can be requested. A download/upload bandwidth limit is not yet implemented, but would also be managed by this component. Components can register as “transmission listener” to be informed about new downloads/uploads and changes of their status.

6.4 P-Grid Package

The P-Grid package is a subpackage of the Server Package. The main `PGrid` class handles all parameters of the P-Grid network: the local path, the references to other hosts for each level of the path, and locally managed files. Initially these files are the files managed by the File Manager. To reduce the complexity of `PGrid` essential processes are handled by subclasses, as shown in Figure 6.2.

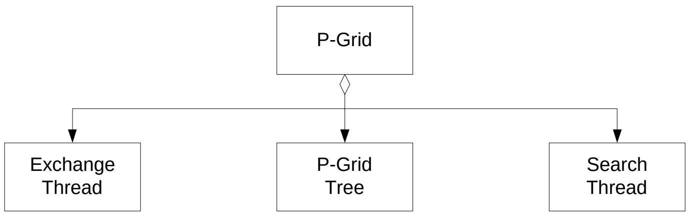


Figure 6.2: The P-Grid architecture

The **Exchange Thread** class (`ExchangeThread`) performs the P-Grid construction algorithm described in Section 3.2.

The **P-Grid Tree** class (`PGridTree`) performs the mapping of filenames into binary strings described in Section 3.3.

The **Search Thread** class (`SearchThread`) performs the P-Grid search algorithm described in Section 3.1.1.

6.5 Communication Package

The Communication package includes all components required for communicating with other hosts. It is a subpackage of the Server package and is also used directly by the P-Grid package. The class diagram is given in Figure 6.3.

The **Communication Acceptor** (`CommAcceptor`) is started by the Communication Manager for each incoming connection request of another host. The Acceptor tries to decide the requested protocol, accepts the connection and returns the Connection to the Communication Manager.

The **Communication Connector** (`CommConnector`) is started by the Communication Manager to establish a connection to another host. The requested

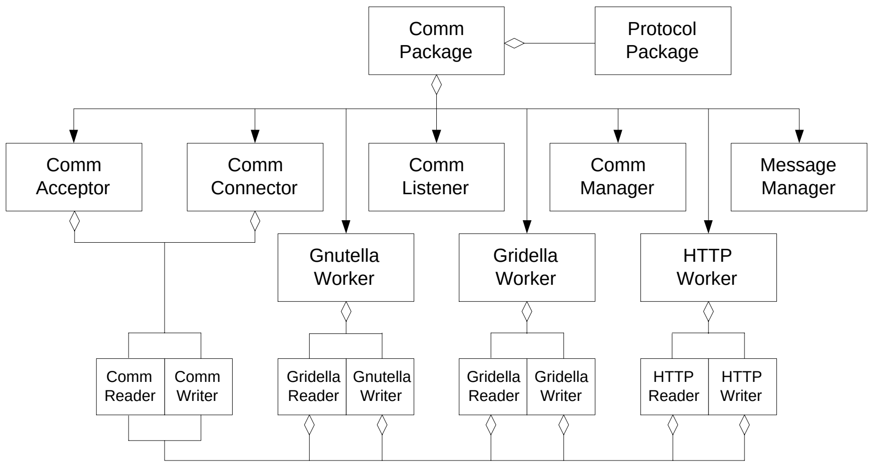


Figure 6.3: The Communication architecture

protocol is given by the Connection component. After protocol handshake the Connection is returned to the Communication Manager.

The **Communication Listener** (`CommListener`) is started at system startup and listens at the given default port to accept incoming connections. Every connection is then handled by the Communication Manager.

The **Communication Manager** component (`CommManager`) handles all connection requests and starts the corresponding workers (Acceptor or Connector). Successfully established connections are then processed by the corresponding workers (Gnutella, Gridella, or HTTP). Connections are administrated and can be accessed to monitor the network status.

The **Message Manager** component (`MsgManager`) administrates all received and sent Gnutella messages. All messages are stored to be able to route back response messages of other hosts. Moreover they are processed locally, i.e., search requests are forwarded to the File Manager. Received response messages for previously received request messages and locally created messages (e.g., Pings or Queries) are added to the send queue of the corresponding target Connection (see 6.2). Components can register as "message listener" to be informed when new messages are received. This can be used to monitor the network traffic.

The **Communication Reader** (`CommReader`) provides basic functionality to read messages from its given connection. It understands Gnutella, Gridella, and HTTP messages. The **Gnutella Reader** (`GnutellaReader`) uses the Communication Reader to read Gnutella messages and adds these to the receiving queue of the corresponding Connection. The **Gridella Reader** (`GridellaReader`) uses the Communication Reader to read Gridella messages and adds these to the receiving queue of the corresponding Connection. The **HTTP Reader** (`HTTPReader`) uses the Communication Reader to read HTTP messages and adds these to the receiving queue of the corresponding Connection.

The **Communication Writer** (`CommWriter`) provides basic functionality to write messages to its given connection. It understands Gnutella, Gridella, and HTTP messages. The **Gnutella Writer** (`GnutellaWriter`) uses the Communication Writer to write Gnutella messages taken from the send queue of the corresponding Connection. The **Gridella Writer** (`GridellaWriter`) uses the Communication Writer to write Gridella messages taken from the send queue of the corresponding Connection. The **HTTP Writer** (`HTTPWriter`) uses the Communication Writer to write HTTP messages taken from the send queue of the corresponding Connection.

The **Gnutella Worker** (`GnutellaWorker`) transforms the received Gnutella messages by the receiving queue into general data structures and delivers this to the Message manager for further processing. The Message manager adds generated response messages autonomously to the send queue of the corresponding Connection.

The **Gridella Worker** (`GridellaWorker`) transforms the received Gridella messages by the receiving queue into general data structures and delivers this to the P-Grid component for further processing. The P-Grid component adds generated response messages autonomously to the send queue of the corresponding Connection.

The **HTTP Worker** (`HTTPWorker`) processes “pushed” downloads from other hosts and upload requests which were forwarded to the Transmission manager.

6.6 Protocol Package

The Protocol Package supports all message types of the Gnutella, the Gridella, and the HTTP protocol. This package is a subpackage of the Communication package and all classes are also only handled by classes of the Communication package. This allows to add new protocols without interfering in too many classes.

6.6.1 Gnutella messages

The Gnutella message types are shown in Figure 6.4.

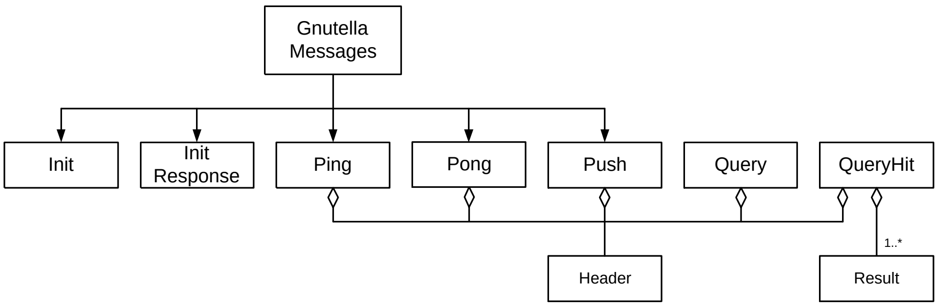


Figure 6.4: The Gnutella Protocol Messages

The **Init message** (`GnutMsgInit`) is sent to establish a Gnutella connection.
The **Init Response message** (`GnutMsgInitResp`) is sent as response to a init message if the requested Gnutella connection is accepted.
The **Ping message** (`GnutMsgPing`) is sent to detect other Gnutella hosts.
The **Pong message** (`GnutMsgPong`) is sent as response to a received Ping message.
The **Push message** (`GnutMsgPush`) is sent from a host which wants to download a file from a firewalled host. The firewalled host is then "pushing" the file to the requesting host.
The **Query message** (`GnutMsgQuery`) represents a search request.
The **Query Hit message** (`GnutMsgQueryHit`) is sent as response to a Query message and contains information about the found files matching the query string at a host. Information (file name, file size, file index, ...) about the files themselves are provided in the message **Result** part (`GnutMsgQueryHitResult`).
The Ping, Pong, Push, Query and Query Hit messages further contain a leading **Header** (`GnutMsgHeader`) holding information about the message. All message classes implement the **Gnutella Message Interface** (`GnutMsgInterface`).
For further details about the Gnutella messages see the Gnutella protocol specification [9] and Section 2.2.

6.6.2 Gridella messages

The Gridella message types are shown in Figure 6.5.

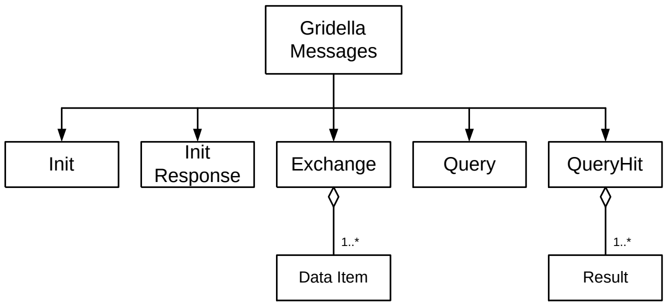


Figure 6.5: The Gridella Protocol Messages

The **Init message** (`GridMsgInit`) is sent to establish a Gridella connection. The **Init Response message** (`GridMsgInitResp`) is sent in response to an init message if the requested Gridella connection is accepted.
The **Exchange message** (`GridMsgExchange`) is sent if two peers meet and used to maintain the Gridella structure. It contains information about the path and the references of the sending host. The **Data Item** part (`GridMsgExchangeDataItem`) holds the data items managed by the sending host.
The **Query message** (`GridMsgQuery`) is sent to search for a query string.
The **Query Hit message** ((`GridMsgQueryHit`) is sent in response to a Query message and contains information about found files matching the query string at a host. Information (file name, file size, file index, ...) about the files themselves are provided in the message **Result** part (`GridMsgQueryHitResult`).
All message classes implement the **Gridella Message Interface** (`GridMsgInterface`).
For further details about the Gridella messages see the Gridella protocol specification given in Section 5.2.

6.6.3 HTTP messages

The HTTP messages are shown in Figure 6.6.

The **GET message** (`HTTPMsgGet`) is sent by a host requesting to download a file from another host. The message contains the file index (unique at a host, given by the File manager to identify a file), the filename and the range of bytes requested.
The **GIV message** (`HTTPMsgGiv`) is sent by a host in response to a received Gnutella Push message, and starts the pushed download. The message contains

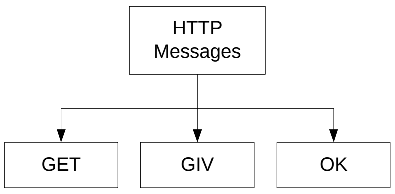


Figure 6.6: The HTTP Protocol Messages

the requested file index (unique at a host, given by the File manager to identify a file), the servent identifier of the Gnutella Push message, and the filename.
The **OK message** (`HTTPMsgOK`) is sent as response to a received GET message and is followed by the requested file.
For further details about the HTTP messages see the message specifications in Appendix D.

6.7 A typical Interaction

To illustrate the dependencies between the described packages a user search request is shown in the following sequence diagrams. Figure 6.7 shows the processing in the Server and P-Grid packages. The search query was entered at the GUI, which called the `addSearch(query)` method of the Client component and which in turn forwarded the request from the Client to the Server component. As shown in Figure 6.7 the Server then starts a search in the Gnutella network by adding the query to the Message Manager as well as calling the `addSearch(query)` method of the P-Grid component to start the search in the Gridella network. The Message Manager then requests a list of all currently connected Gnutella hosts to add the query to the send queues of these connections. For a Gridella search the P-Grid component starts a dedicated SearchThread for each word of the query string. This is necessary because P-Grid only supports prefix matches of query strings. To receive not only files beginning with the first entered key string, the query is divided into several queries containing only one of the entered key strings. The user can define this behavior via the GUI.

A search in the Gnutella network is only performed at currently connected hosts. A search in the Gridella network can lead to the necessary to connect a Gridella host to process the search request successfully (shown in Figure 6.8).

First the SearchThread finds the responsible host for the query and requests a connection to that host from the Communication Manager. If the host is already

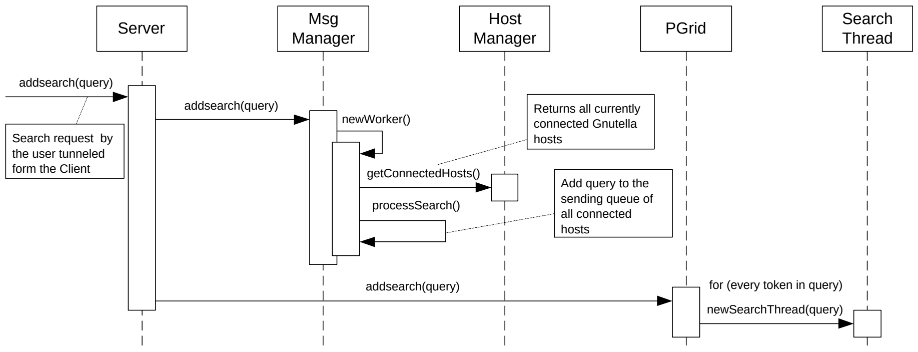

Server
Msg Manager
Host Manager
PGrid
Search Thread
addsearch(query)
Search request by the user tunneled form the Client
addsearch(query)
newWorker()
Returns all currently connected Gnutella hosts
getConnectedHosts()
processSearch()
Add query to the sending queue of all connected hosts
addsearch(query)
for (every token in query)
newSearchThread(query)

Figure 6.7: Sequence diagram for a user search requests

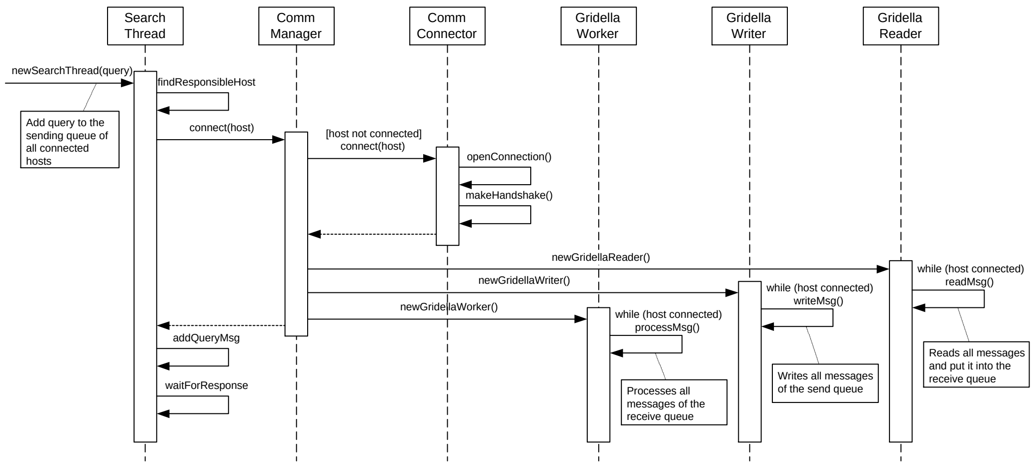

Search Thread
Comm Manager
Comm Connector
Gridella Worker
Gridella Writer
Gridella Reader
newSearchThread(query)
findResponsibleHost
Add query to the sending queue of all connected hosts
connect(host)
[host not connected] connect(host)
openConnection()
makeHandshake()
newGridellaReader()
while (host connected) readMsg()
newGridellaWriter()
while (host connected) writeMsg()
newGridellaWorker()
while (host connected) processMsg()
addQueryMsg
waitForResponse
Reads all messages and put it into the receive queue
Writes all messages of the send queue
Processes all messages of the receive queue

Figure 6.8: Establishing a Gridella connection

connected with the Gridella protocol, this connection is returned. Otherwise a new connection is established and returned. The new connection is established by the CommConnector, which opens the connection and makes the protocol handshake. After successfully establishing the Gridella connection a worker for reading (Gridella Reader) and writing (Gridella Writer) messages from/to the host is created and the connection is returned to the SearchThread. The reader waits for incoming messages and puts them into the receive queue and the writer sends all messages of the send queue as long as the host is still connected. The started Gridella worker waits for new messages in the receive queue and performs them. The SearchThread puts its Query message into the send queue, and waits for a QueryReply message forwarded by the Gridella Worker. If the QueryReply message is received the Gridella Worker forwards the message to the waiting SearchThread which now continuous processing.

Chapter 7

Implementation

This chapter describes the Java implementation of the design presented in the previous chapter. The design has been mapped onto six Java packages:

- `gridella.Client` contains the Client component for interfacing with the Server, and a player to open files directly from the application.
- `gridella.general` contains a configuration facility, utilities and general data structures which can be accessed and used by all other classes.
- `gridella.Server` contains the Server component to enable the Client to interface the Server. To administrate all hosts the package contains a Host Manager, a File Manager to administrate the shared files, and a Transmission Manager for all downloads/uploads.
- `gridella.Server.PGrid` contains the P-Grid component and all its subclasses to enable the full P-Grid functionality.
- `gridella.Server.Comm` contains communication components like managers for connections and messages, workers for all types of supported protocols and message writers and readers.
- `gridella.Server.Comm.Protocol` contains the message classes encapsulating the supported protocols.

The remainder of this chapter discusses the implementations of the individual packages.

7.1 Client Package

The `Client` class provides methods to interface with and monitor the Server by forwarding all method calls to the corresponding methods of the Server. A Graphical User Interface (GUI) as included in the implementation should only call these methods.
The `Player` class is used to open local files with applications defined by the user.

7.2 General Package

The configuration facility (`Cfg` class) provides global constants and access to all configuration parameters, that were read in at startup from the property file (`gridella.properties`) described in Appendix A.
A connection is represented by the (`Connection`) class. The class uses the `LinkedQueue` class of the `EDU.oswego.cs.dl.util.concurrent` package to realize the send and receive queue used by several classes, possibly concurrent. The package contains a utility class (`Utils`) which provides useful methods to convert hexadecimal strings to bytes and back, formating and tokenizing of strings. Another utility is the XML parser (`XMLParser`). It is initialized with a string containing a XML document, e.g., a received Gridella message. The parser is then able to extract all values of the XML document. In Figure 7.1 the XML parser is created with a string representing a received Gridella Query message from which the *Key* and the *Query* values are needed for further processing.

```
...
String xmlDoc = new String("<Gridella Version=\"1.0\">
                              <Host IP=\"128.132.127.24\" Port=1805/>
                              <Query Index=3 Key=010100011>Madonna Rain</Query>
                            </Gridella>");
XMLParser parser = new XMLParser(xmlDoc);
String key = parser.getValue("Gridella/Query/@Key");
String query = parser.getValue("Gridella/Query/.");
...
```

Figure 7.1: Using the XML Parser

Furthermore the package contains data structures used by other classes to exchange information between the different classes and packages. For example the `GUID` class represents a Global Unique Identifier (GUID), which is used in the Gnutella and the Gridella protocols to uniquely identify hosts and messages. The identifier is created by the (`GUIDGenerator`) class, shown in Figure 7.2. The generator can be created with a given digestion algorithm or the default (SHA) is set. The local Internet address (host name/ip address) is used to create a global unique string. Then a string containing the current date and time is appended followed by a random number. The `MessageDigest` class of the `java.security` package is then used with the specified digestion algorithm and the generated string to generate a new GUID.

7.3 Server Package

The `Server` class represents the interface to the server for all other classes outside the Server package and forwards all requests to the responsible classes.
The File Manager (`FileManager`) manages all local shared files and therefore is responsible for all incoming Gnutella search requests. Each request is processed by a new thread (the relevant code is shown in Figure 7.3). The thread is executed by the `PooledExecutor` class of the package `EDU.oswego.cs.dl.util.concurrent`, and uses the regular expression utility of GNU to find all matches between the query and the local filenames summarized in the string *mFileNames*. The matching files are then delivered to the Message Manager to create the response message.
The Host Manager (`HostManager`) fulfills administrative functions to maintain lists of Gnutella and Gridella hosts, and to keep up Gnutella connections to the configured number of.
The Transmission Manager (`TransManager`) administers all locally started downloads and all uploads from the local host. It provides not only functions to start and stop them, but also to restart a download at the aborted position to avoid a multiple download.

7.4 P-Grid Package

The `PGrid` class handles all parameters of the P-Grid network: local path, references to other hosts, and files managed by the local host. All these data are also stored in local files (see Appendix B and Appendix C) and read in at startup. Further it is responsible to keep these information up-to-date by performing Exchanges with other peers. Exchanges and search requests are executed by the `PooledExecutor` class of the package `EDU.oswego.cs.dl.util.concurrent` in

```java
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public class GUIDGenerator {

  ...

  /**
   * Creates a unique ID generator and initializes it. The default
   * digestion algorithm is set to SHA.
   */
  public GUIDGenerator() {
    algorithm = "SHA";
  }

  /**
   * Called by the public generation methods; generates the
   * seed and digests it (using the algorithm defined in
   * this.algorithm); the digest is the unique ID.<BR>
   * The seed is a random string of the form
   * <host>/<ip adress><date><random number> and generated
   * new with every call.
   * @return the newly created uniqueID.
   * @throws NoSuchAlgorithmException if the algorithm is
   *         not available in the caller's environment.
   */
  private GUID _generate() throws NoSuchAlgorithmException {
    try {
      seed = InetAddress.getLocalHost().toString();
    } catch (UnknownHostException e) {
      seed = "localhost/127.0.0.1";
    } catch (SecurityException e) {
      seed = "localhost/127.0.0.1";
    }
    seed = seed + new Date().toString();
    seed = seed + new Long(new Random(new Date().getTime()).nextLong()).toString();
    MessageDigest md = MessageDigest.getInstance(algorithm);
    md.update(seed.getBytes());
    GUID uid = new GUID(md.digest());
    return uid;
  }
  ...
```

Figure 7.2: Global Unique Identifier generation

```
import gnu.regexp.*;
...
/**
 * Constructs a new thread to process the new query.
 * @param query the new query to process.
 */
public processQuery(Query query) {
  mQuery = query;
}

/**
 * Starts the processing thread.
 */
public void run() {
  if (mQuery.getSearchStr().trim().length() == 0)
    return;                    // empty query
  if (mQuery.getMinSpeed() > mCfg.getResourceInt("ConnectionSpeed"))
    return;                    // higher connection speed than local required

  String[] keys = Utils.tokenize(mQuery.getSearchStr().trim().toLowerCase());
  Vector foundFiles = new Vector();

  String pattern = new String();
  for (int i = 0; i < keys.length; i++) {
    pattern = pattern.concat("([^\n]*" + keys[i] + "[^\n]*)");
    if ((i+1) < keys.length)
      pattern = pattern.concat("|");
  }
  RE regexp = null;
  try {
    regexp = new RE(pattern, RE.REG_ICASE);
  } catch (REException e) {
    e.printStackTrace();
    return;
  }
  REMatch[] matches = regexp.getAllMatches(mFileNames);
  for (int i = 0; i < matches.length; i++)
    foundFiles.add(mFilesByName.get(matches[i].toString()));

  if (foundFiles.size() > 0)
    mMsgMgr.queryResponse(mQuery, foundFiles);
}
```

Figure 7.3: Find matches between local files and query string

a separate thread.
The P-Grid search requires a mapping of query strings into binary strings, which is performed by the `PGridTree` class. To achieve a uniformly distributed matching of query strings to binary strings a balanced trie structure based on a sample database of search strings is used for this computation (see Section 3.3. To speed up startup, this trie is stored in binary format to a local file and which is used at the next startup.
To construct and maintain the P-Grid exchanges are done periodically by the `ExchangeThread`. The thread is either started by the `PGrid` class or because an Exchange message was received from another host. In the first case, an Exchange message is sent to the host to exchange and the thread waits for the response, in the second case, an Exchange message is sent as response immediately. The exchange algorithm described in Section 3.2 may extend the local path by one or more random bits created by the method shown in Figure 7.4. The method on the other host would return a string of the inverse bits.

If a search request is defined by the user or received from another host, a dedicated thread (`SearchThread`) is executed for each request. The local path and the mapped query string are used to determine the responsible host for these query. If the local host is responsible, it uses the regular expression utility by GNU as already described in Section 7.3 to find matches with the locally managed files. If another host is responsible, the query is forwarded and the thread waits for a response to arrive.

7.5 Communication Package

To accept incoming connections the `CommListener` class listens at the configured port (1805 by default) and the connection is then handled by the `CommManager`. The `CommManager` starts an `CommAcceptor` for each connection to decide the requested protocol and to perform the protocol handshake. Depending on the decided protocol, the corresponding worker, reader and writer are started. The `CommManager` can also be used to connect a host by starting a `CommConnector` to establish the connection and to perform the protocol handshake.

The `CommReader` class is used by all protocol readers to read messages from the input stream and the `CommWriter` class to write messages to the output stream. The `GnutellaReader` uses the `CommReader` to receive Gnutella messages and puts them into the receive queue of this connection. The `GnutellaWriter` uses the `CommWriter` to send Gnutella messages taken from the send queue of this connection.

```java
/**
 * Returns the path extension for this host.
 * @param locally flag if the exchange was started by the local host or remote.
 * @param bit the amount of bits.
 * @return the path extension.
 */
private String pathExtension(boolean locally, int bits) {
  byte[] lAddr = mCfg.getLocalHost().getAddr().getAddress(); // the local address
  byte[] rAddr = mExchange.getHost().getAddr().getAddress(); // the remote address
  long val = mCfg.getLocalHost().getPort();                  // the local port
  val += mExchange.getHost().getPort()+3;                    // the remote port
  // add the bits of the Internet addresses
  for (int i = 0; i < lAddr.length; i++)
    val += (int)lAddr[i];
  for (int i = 0; i < rAddr.length; i++)
    val += (int)rAddr[i];
  String rev;
  if (locally) {
    // exchange was started locally => use value
    rev = Long.toBinaryString(val);
  } else {
    // exchange was started remote => use the inverse value
    rev = Long.toBinaryString(~val);
    // remove the leading '1' of negative values
    rev = rev.substring(rev.length() - Long.toBinaryString(val).length());
  }
  String binary = new String();
  // create binary string starting with LSB of rev
  for (int i = rev.length()-1; i >= 0; i--)
    binary = binary.concat(rev.substring(i, i+1));
  // append the string by the requested length
  while (binary.length() < bits) {
    binary = binary.concat(binary);
  }
  return binary.substring(0, bits);
}
```

Figure 7.4: Find a string of bits to extend the path

The GridellaReader uses the CommReader to receive Gridella messages and puts them into the receive queue of this connection. The GridellaWriter uses the CommWriter to send Gridella messages taken from the send queue of this connection.
The HTTPReader uses the CommReader to receive HTTP messages and puts them into the receive queue of this connection. The HTTPWriter uses the CommWriter to send HTTP messages taken from the send queue of this connection.

The GnutellaWorker class handles all messages received of a Gnutella connection. To handle all messages in time the worker can start further workers if the amount of messages in the receive queue exceeds a certain threshold. These threads are executed by the PooledExecutor class of the package EDU.oswego.cs.dl.util.concurrent. Each worker handles a message, checks its validity and adds it to the Message Manager for further processing.
The GridellaWorker class handles all messages received of a Gridella connection. It checks the messages for validity and transforms them into a general data structure to forward the request to the PGrid class for further processing.
The HTTPWorker handles all messages received of a HTTP connection and therefore is responsible for accepting upload and Gnutella Push requests, which will be forwarded to the Transmission Manager.

The Message Manager (MsgManager) administers all received and sent Gnutella messages, forwards received messages to the corresponding hosts or multicasts them to all connected hosts if required. Incoming Ping requests are processed directly by the Message Manager, while search requests are forwarded to the File Manager. Messages to be sent are simply added to the send queue of a connection.

7.6 Protocol Package

This package includes classes representing messages of all supported protocols.
All Gnutella message classes implement the GnutMsgInterface interface. The greeting message (GnutMsgInit) is sent after establishing the connection, and GnutMsgInitResp is sent in response. All other messages contain a header represented by the GnutMsgHeader class. Ping messages (GnutMsgPing) are used to detect new hosts in the network which respond with a Pong message (GnutMsgPong). A Query Hit message (GnutMsgQueryHit) is sent as response to a Query message (GnutMsgQuery) and contains one or more instances of class GnutMsgQueryHitResult representing a file matching the search criteria.

All Gridella message classes implement the `GridMsgInterface` interface. The greeting message (`GridMsgInit`) is sent after establishing the connection, and `GridMsgInitResp` is sent in response. Exchange messages (`GridMsgExchange`) can contain one or more instances of class `GridMsgExchangeDataItem` representing a file managed by the sending peer. A Query Hit message (`GridMsgQueryHit`) is sent in response to a Query message (`GridMsgQuery`) and contains one or more instances of class `GridMsgQueryHitResult` representing a file matching the search criteria.

A HTTP GET request (`HttpMsgGet`) is used to download a file from a remote host. Therefore the filename as well as the file index are required. To support resuming of aborted downloads the byte range to download can also be defined. If a direct download is not possible, because the requested host is behind a firewall, the Gnutella protocol supports pushed downloads. A host receiving a Gnutella Push message will open a connection to the host requesting the download and send the HTTP GIV message (`HttpMsgGiv`) to initialize the pushed download. If a download request is accepted a HTTP OK message (`HttpMsgOK`) is returned followed by the requested data.

Chapter 8

Related Work

A number of approaches are described in the literature that address scalable, decentralized access schemes. All of them differ from our approach because they exclusively allow exact search for file (or object) identifiers rather than text-based search for filenames. Also the autonomy of peers is normally severely limited with respect to the kind of search requests that they have to support whereas our approach supports considerable freedom. Part of the related work was already presented in Chapter 2 where current Peer-to-Peer systems like Gnutella were discussed.

A prototype Query/Advertise system called Quad [17] was developed at the University of Coloroda using the existing Java-based prototype of the Siena [8] publish/subscribe middleware to provide a canonical architecture. A client acting as a resource provider describes his available resources using a filter pattern and establishes a subscription (advertisement) based on that filter at the nearest server. The server then forwards the subscription filter to all of its peers. Each peer notes where the subscription came from, and forwards it to its peers. Later, some other clients may construct a message describing the resource it is interested in, inserts the message into the Q/A system and the local copy of the filter is applied at the contacted server to determine the next server to which the message should be forwarded. If no filter matches at the local server, then it will not be forwarded and so no inter-server traffic will be generated. Each advertiser that receives a message must perform a detailed examination of its resources and construct a message describing each matching resource in more detail. This response is then routed back to the query originator. The response must include some identification for the matching advertiser so that the query originator can obtain the actual resource. The amount of filters a server must maintain is

reduced by the “Covers” relation over filters. If one filter *covers* another filter, only the more general filter must be stored.
Advantages of this system are the anonymity of the users, because only one server needs to know the IP address of a client. The use of advertisements and the “Covers” relation results in low network traffic and replies can also be routed back along the path from advertiser to the query originator. Malformed messages from a client and clients trying to flood the network can be caught and suppressed by the servers.
The main disadvantage of this system is the separation of clients and servers and therefore it is impossible to dynamically extend the network.

OceanStore [26] is a distributed, fault-tolerant, secure storage infrastructure which was designed for billions of users and exabytes of data. It consists of millions of individual servers, each cooperating to provide service. A group of such servers is a pool. Data flows freely between these pools, allowing replicas of a data object to exist anywhere, at any time. Because OceanStore is composed of untrusted servers, it utilizes redundancy and client-side cryptographic techniques to protect data. Although many servers may be corrupted or compromised at a given time, the complete systems aggregate behavior assures users of stable storage. Moreover, because client data is encrypted, servers are never able to read it. Users are assumed to pay for service from one of many possible OceanStore service providers (OSPs), each of which own some of the OceanStore servers. Input data is split into fragments and spreaded over many servers, whereas only a fraction of the fragments is needed to reconstruct the original block. Each object is identified by a Global Unique Identifier (GUID) which is the result of a secure hash function over the object’s content. To locate enough servers storing fragments of a object, an overlaying routing and location layer (Tapestry) on top of TCP/IP that maps GUIDs to individual servers is used. It uses local neighbor maps to incrementally route messages to their destination, digit by digit, which is similar to the routing scheme used by Gridella.
As OceanStore is designed as data storage system for “private” data, it does not support search algorithms or data sharing for all users.

Freenet [13] is a file sharing application that allows files to be inserted, stored and requested, while protecting the anonymity of both authors and readers. Each node contains information about keys (that are analogous to URLs) of other Freenet nodes and the data addressed by those keys. When a node receives a query to which it cannot respond, it forwards the query to the node that knows about keys that are close to the requested query by using a key similarity measure. If a node receives a query to which it can respond, it sends the data back to the requesting node in the chain. As responses are routed back along the chain of nodes to the original requester, each node replicates the data. If cached data is not requested for some period of time it is discarded. From a system point of view Freenet probably is the approach closest to ours although it supports only searches for file identifiers.

Chord [10] is a decentralized P2P lookup service that stores key/value pairs for distributed data items. Given a key, the node responsible for storing the key's value can be determined using a hash function that assigns an identifier to each node and to each key (by hashing the node's IP address and the key). Each key k is stored on the first node whose identifier id is equal or follows k in the identifier space. Each node maintains a routing table with information for only about $O(\log N)$ nodes. With a high probability, the number of nodes that must be contacted to resolve a query in a network of N nodes is $O(\log N)$. Chord cannot be extended to support searches more general than using key identity.

In [25] a distributed data access scheme is proposed that is structurally similar to P-Grid. For building the access structure the method relies on global knowledge, in particular, on the total number of participants. The main emphasis is on determining the most cost-effective access paths, based on a communication cost function and a detailed probabilistic analysis showing that with high probability with $O(\log N)$ cost, requests for key values can be answered with bounded cost. No replication is used in this scheme and only search on key identity is possible.

Mariposa [28] is a distributed DBMS designed for the requirements of wide-area networks (WAN). It uses microeconomic principles, in particular auctions, for managing query execution and storage management to avoid complex global solutions. The approach lacks support for indexed data access.

Terminodes [18] (terminal+node) are mobile devices that provide functionality of both terminals and nodes of the network. A network of terminodes is decentralized, autonomous, self-organizing and independent of any infrastructure. As being a decentralized, self-organizing networking infrastructure it can be used to implement self-organizing applications on top of it which is part of our future work.

Besides these related P2P approaches there is also work going on to increase the interoperability of P2P systems and define a universal architecture for P2P systems. Project JXTA [15] attempts to provide a network-programming platform for P2P systems. It defines a three layer P2P software architecture, a set of six XML-based protocols, and a number of abstractions and concepts such as peer groups, pipes, and advertisements to provide a uniform platform for applications using P2P technology and for various P2P systems to interact. We are evaluating it for Gridella.

Chapter 9

Evaluation and Future Work

9.1 Evaluation

Gridella can be viewed as a layer on top of Gnutella that provides additional abstractions and functionality: Gnutella provides the basic communication and Gridella adds directed, efficient search due to its underlying P-Grid approach. This is superior to Gnutella which searches in a trial-and-error way and thus requires considerably more network bandwidth. In an analytic performance study the number of messages required to find a specific data item in Gridella and Gnutella was compared. Peer populations between 20.000 and 200.000 were considered and peers that are online with a probability of 0.3. The goal was to achieve a search success probability of 0.99. For Gridella it was assumed that each peer stores 1000 data items which is necessary to determine the key length. To achieve a search success probability of 0.99 in Gnutella it is sufficient to create 22 replicas of each data item and to have a search horizon of 70% of all Gnutella peers, i.e., to reach 70% of the Gnutella population with a search message. Based on this the required number of search messages (see the formula in Section 2.2.1) can be computed. For Gridella the number of messages required to traverse the Gridella search structure in the worst case was determined. The results are shown in Table 9.1.

The variations for Gridella are the result of the simulation of the probabilistic process simulating Gridella's search algorithm. The steps in the results for Gnutella correspond to the steps in the time-to-live which must be incremented for higher numbers of peers to meet the 70% requirement. The results clearly

Peers	**Gridella messages**	**Gnutella messages**
20000	61	8744
40000	63	26240
60000	65	26240
80000	65	78728
100000	68	78728
120000	69	78728
140000	68	78728
160000	69	78728
180000	69	78728
200000	72	78728

Table 9.1: Performance comparison of Gridella and Gnutella

demonstrate the benefit of using an access structure even if we have to take into account some modest storage demand and update overhead in Gridella.

As has been shown in [1] P-Grid offers several additional advantages which can be exploited by Gridella: It provides analytical proofs on bounds for reliability of answering search requests, fault-tolerance is high, and the structure of the overall system can be configured flexibly.

Gridella has been developed in a modular architecture, which allows the replacement and expansion of any component. The creation of the P-Grid network was tested successfully, which allowed the users to find and download all shared files of the network. The used protocol for Gridella is XML-based and requires no parts of the Gnutella protocol and is open for further protocols to be added. Gridella can communicate with other Gridella hosts (hosts that support only the Gridella protocol) but also with “normal” Gnutella hosts. It is therefore able to infiltrate the existing Gnutella network. The GUI allows the user to use all functions available from the Gridella and Gnutella network, as well as monitoring all network traffic.

9.2 Future Work

Experiments with the current implementation have shown some potential for further improvements in structure and efficiency. We plan to release an out-of-the-box set of Gridella components which can be included in other systems or enriched with new GUIs.

The included GUI could also be extended by further functions, like searching in search results, specify a maximum file size, search only for specified file extensions or group the search results by filenames. To enhance the performance of downloads, identical files on different hosts could be identified to support simultaneous downloads of parts of a file from different hosts.

Conceptually part our future work will concentrate on the inclusion of reputation and trust in Gridella based on the approach proposed in [3]. This approach also relies on P-Grid so that the existing implementation can be reused in multiple ways. The natural next step then would be to address security issues such as authenticity and confidentiality of information. With these improvements in place P2P would also be a very interesting new environment for e-commerce. Additionally we plan to add substring search and support for structured, semantically self-organized search [4].

As IP addresses have become a scarce resource most computers on the Internet no longer have permanent addresses. In P-Grid ad-hoc connections to peers have to be established, which can only be done if the receiving peer has a permanent IP address. We plan to realize an approach [16] for a completely decentralized, self-maintaining, light-weight, and sufficiently secure peer identification service that allows to consistently map unique peer identifications onto dynamic IP addresses in environments with low online probability of the peers constituting the service. For security a combination of PGP-like public key distribution and quorum-based query scheme is applied.

To enrich the availability and performance of the P-Grid network, research must be done to handle all kinds of attacks like DoS attacks or byzantine hosts. Methods must be found to detect the attack and algorithms to keep the network alive, even when a whole part of the network is unusable. Replication will not last out to avoid this.

Also we intend to apply concepts to remedy the free-riding problem. A possible solution could be the introduction of economic concepts, where users have to "pay" for the services they use. By paying we do not necessarily mean the exchange of monetary values but a more market-driven approach where a micropayment system with an artificial currency could be used to balance requests with offers as suggested by [23]. Offers and downloads from a peer would earn the peer credits which it in turn could use for paying for services it requests from other peers.

Chapter 10

Conclusions

By popularizing the P2P approach in simple yet very successful and influential systems such as Napster and Gnutella the "Internet Community" has proven again its incubator capabilities for revolutionary systems and has somewhat "outperformed" the scientific community. A similar thing has occurred some time ago with the introduction of the World-wide Web which boosted the Internet into everyday life. Although such developments are very helpful for spreading and advancing new technologies we know now that this often comes at the cost of lacking scientific foundations which impedes development in the long run.

At the moment we still have the opportunity to put P2P systems on firm scientific foundations by combining state-of-the-art methodological and engineering know-how and thus advancing this paradigm. The work presented in this thesis is a first step in this direction. It improves the highly chaotic and inefficient Gnutella infrastructure with directed search and other advanced concepts which enhance efficiency and provides a consistent mathematical model for further reasoning and research. Still a considerable amount of research and experiments will be necessary to make P2P systems feasible for application domains beyond mere MP3 and image exchange, for example as a new paradigm for decentralized e-commerce systems.

To reach the long-term goal to replace the existing Gnutella infrastructure by the P-Grid infrastructure it is necessary to evolve the implemented prototype application to deliver a competitive alternative to the existing applications like KaZaA, Grokster, Morpheus, LimeWire or BearShare. A better infrastructure leads not automatically to an high number of users and thereby not to an higher number of available data items. The modular architecture allows the replacement of certain parts of the application, so that a better GUI, additional functionality, or further protocols can be exchanged or extended easily.

Appendix A

Sample Gridella Configuration File

The configuration of Gridella is done via the `gridella.properties` file in the application directory. It can be opened and edited with a text editor. The file is structured in several parts and each property is represented by a `name=value` pair. To guarantee the correct work of the application, only the properties described in this manual should be changed and only the described values should be used.

A.1 General settings

1. **Look & Feel**: Sets the standard Look & Feel for the application. Possible values for the property `Look&Feel` are: `mLook&Feel.windows`, `mLook&Feel.metal` and `mLook&Feel.motif`.
   Default: `mLook&Feel.metal`

2. **Window Size**: The size of the application window can be set at `SizeX` and `SizeY`.
   Default: `SizeX=700`, `SizeY=600`

3. **P-Grid configuration file**: The P-Grid configuration file includes information about the Gridella network. The file can be set at `CfgFile`. If the given file does not exist a new file will be constructed.
   Default: `PGridHosts.ini`

4. **Minimal storage**: This value indicates the maximal number of file references Gridella administers. `MinStorage` can be any valid integer value. The higher the value the more files are stored and administered.
   Default: `100`

5. **P-Grid files**: The local files are stored in the `GridFiles` file. If this file does not exists, a new one is created at startup.
   Default: `PGridFiles.ini`

A.2 Download

1. **Temporary directory**: Sets the temporary directory for all downloads. `TempDir` should be an existing directory or a directory that can be created at startup. Use forward slashes ('/') instead of backslashes ('\')!
   Default: `Temp/`

2. **Download directory**: Sets the download directory for all downloads. `DownloadsDir` should be an existing directory or a directory that can be created at startup. Use forward slashes ('/') instead of backslashes ('\')!
   Default: `Downloads/`

A.3 Network

1. **Gnutella hosts**: Gnutella service hosts are required to enter the Gnutella network. A list of this hosts can be given at `GnutHosts`. The hosts should be represented by their IP-address or their name and the listening port ($< ip : port >$ or $< name : port >$). The hosts should be separated by whitespace.
   Default: `gnutellahosts.com:6346 router.limewire.com:6346 gnutella.hostscache.com:6346 connect1.bearshare.net:6346 connect2.bearshare.net:6346 connect3.bearshare.net:6346 connect1.gnutellanet.com:6346 connect2.gnutellanet.com:6346 connect3.gnutellanet.com:6346 connect4.gnutellanet.com:6346`

2. **Gridella hosts**: Gridella service hosts are required to enter the Gridella network. A list of this hosts can be given at `GridHosts`. The hosts should be represented by their IP-address or their name and the listening port ($< ip : port >$ or $< name : port >$). The hosts should be separated by

whitespace.
Default: `www.p-grid.org:1805`

3. **Connection speed**: The maximum speed this application can communicate with. `ConnectionSpeed` should be a valid integer representing the speed in kb/second.
   Default: `56`

4. **Firewall**: If this host is behind a firewall, the `BehindFirewall` property must be set to *true*, *false* otherwise.
   Default: `false`

A.4 Library

1. **Shared directories**: Sets the shared directories. Files in these directories can be found and downloaded by other users. `SharedDirs` should be a list of existing directories separated by ';'. Use forward slashes ('/') instead of backslashes ('\')!
   Default: `empty`

2. **Include sub-directories**: If all sub-directories of the shared directories should also be shared, the `IncludeSubDirs` property should be set to *true*, *false* otherwise.
   Default: `true`

3. **Shared file-extensions**: To share only files with certain file-extensions a list of these extensions could be given via `SharedExt`. The extensions should be separated by whitespace.
   Default: `mp3 wma mpg mpeg mov avi asf jpg gif rm htm html txt swf wav ram ra qt`

4. **Share all files**: To share all files of any type, the `ShareAllExt` property could be set to *true*. If it is set to *false*, only files with the given file-extensions are shared.
   Default: `false`

A.5 Players

To be able to "play" files directly from the application, the serving applications must be defined. There can be different players for different file types (file

extensions). The players are defined by `Player` followed by the player-index, e.g., the first player is `Player1`, the second `Player2`. The value for the player is the filename of an application. To associate the player with files, the file extensions handled by this player are to be set by `Player?Ext`, where '?' stands for the player-index (1, 2, 3, ...). The list of file extensions should be separated by whitespace.
Default:

```
Player1=C:/Programme/Winamp/Winamp.exe
Player1Ext=m3u asx mp1 mp2 mp3
Player2=C:/Programme/Windows Media Player/mplayer2.exe
Player2Ext=wav snd wma mid midi mpeg mpg m1v mpa mpe mov qt avi
wmf asf wmf wma
Player3=C:/Programme/RealPlayer/realplay.exe
Player3Ext=ra rm ram
```

A.6 Network tab

1. **Auto connect**: If the application should try to connect to the Gnutella network at startup the `AutoConnect` property can be set to *true*, *false* otherwise.
   Default: `true`

2. **Auto cleanup**: If the application should automatically remove illegal or erroneous hosts to reduce resource allocation the `AutoCleanup` property should be set to *true*, *false* otherwise.
   Default: `true`

3. **Maximum to connect**: If the application tries to connect to the Gnutella network automatically the `MaxConnect` property defines the maximum number of hosts to connect concurrently. The value should be a valid integer.
   Default: `10`

A.7 Monitor tab

1. **Show last searches**: The number of the last incoming searches shown can be set via the `ShowLastSearches` property. The value should be a valid integer.
   Default: `11`

2. **Pause show last searches**: If incoming searches should not be shown the property `ShowPause` should be set to *true*, *false* otherwise.
   Default: `true`

Appendix B

P-Grid Routing Table

By default Gridella stores the routing table of P-Grid in the local file `PGridHosts.ini`. The filename can be set in the configuration file, see Appendix A. The syntax of the file is given in Figure B.1:

```
. <Global Unique Identifier> <IP address> <port>
<path[0]> <host[0][0]> <host[0][1]> <host[0][2]> ...
<path[1]> <host[1][0]> <host[1][1]> <host[1][2]> ...
<path[2]> <host[2][0]> <host[2][1]> <host[2][2]> ...
...
* <replica[0]> <replica[1]> <replica[2]> ...
```

Figure B.1: The syntax of the routing table file

The first line begins with a '.' and contains the Global Unique Identifier (GUID), the IP address and the listening port of the local host. The following lines begins with a bit of the local path followed by the references at this level (starting with the most significant bits). The hosts are stored as `<ip:port>`. The last line starts with '*' and contains the replicas of the local host (`<ip:port>`).

Figure B.2 shows an example routing table of a host with the GUID `9E3776F20CC74BACA5983E5627A9186B28CC2B49`, at the IP address `128.157.15.92` using port `1805`. The local path is `101` and there are three hosts at level 0 and two hosts at levels 1 and 2. Host `125.156.150.82:1805` is a replica of the local host.

```
. 9E3776F20CC74BACA5983E5627A9186B28CC2B49 128.157.15.92 1805
1 128.157.13.95:1805 131.157.150.122:1805 120.151.18.2:1805
0 128.127.15.2:1805 123.177.135.56:1805
1 110.137.15.12:1805 112.177.46.34:1805
* 125.156.150.82:1805
```

Figure B.2: Example routing table file

Appendix C

Managed P-Grid Files

Gridella stores the files it manages for the P-Grid network in the local file `PGridFiles.ini`. The syntax of the file is given in Figure C.1:

```
<IP address>TAB<port>TAB<key>TAB<index>TAB<filesize>TAB<filename>TAB<infos>
<IP address>TAB<port>TAB<key>TAB<index>TAB<filesize>TAB<filename>TAB<infos>
<IP address>TAB<port>TAB<key>TAB<index>TAB<filesize>TAB<filename>TAB<infos>
...
```

Figure C.1: The syntax of the P-Grid Files file

Each file is stored in one line, beginning with the IP address followed by a tab stop and the port of the storing host. Then comes the key (the binary representation of the filename), the index of the file on the remote host, the size of the file, and the filename (separated by tabs). If some extra information is available for a file, these information (e.g. bit rate, resolution, track length in seconds, ...) may be appended at the end.

Figure C.2 shows a sample file with three entries. For example, the third file is on the host with the IP address `125.156.150.82` and port `1805`. The key for this file is `00011000` and the index is `3`. The size of the file is `3793209` bytes and the filename is `A-HA - Forever Not Yours.mp3`. There are also extra informations available, i.e., the file's bitrate is `128kbps`.

```
128.157.15.92   1805  11101001  4 4978688 The Calling - Wherever You Will Go.mp3
123.177.135.56  1805  11101001  2 4310168 The Cranberries - Time Is Ticking Out.mp3
125.156.150.82  1805  00011000  3 3793209 A-HA - Forever Not Yours.mp3   128kbps
```

Figure C.2: Example P-Grid Files file

Appendix D

HTTP Messages

HTTP messages are used by the P-Grid network to download files and are identical with the corresponding Gnutella messages (see [9] for detail). Gridella can download from Gridella and Gnutella servents, and accept download requests from Gridella and Gnutella hosts.

D.1 HTTP GET Message

A HTTP GET message is sent by a host requesting to download a file from another host, and can be received and sent at any time. The message contains the `File Index` (unique at a host, assigned by the File manager to identify a file), the `File Name` and the range of bytes requested. The range is given by `Range Begin` and `Range End`. If the whole file is requested, the range is `Range: bytes=0-`. The message syntax is shown in Figure D.1.

Figure D.2 shows an example of a download request. The index of the requested file named `Tori.mp3` is `458`, which allows the receiving peer to identify the file. Only a part of the file is requested as defined in the range attribute.

```
GET /get/<File Index>/<File Name>/ HTTP/1.0\r\n
User-Agent: Gridella\r\n
Referrer: Gnutella\r\n
Connection: Keep-Alive\r\n
Range: bytes=<Range Begin>-<Range End>\r\n
\r\n
```

Figure D.1: HTTP GET message syntax

```
GET /get/458/Tori.mp3/ HTTP/1.1\r\n
User-Agent: Gridella\r\n
Referrer: Gnutella\r\n
Connection: Keep-Alive\r\n
Range: bytes=4678-12487\r\n
\r\n
```

Figure D.2: HTTP GET message example

D.2 HTTP GIV Message

A HTTP GIV message is used by the Gnutella network to support downloads from a firewalled host. It is sent by a host in response to a received Gnutella Push message (see Section 2.2), and starts a pushed download. When a Gnutella peer is not able to download a file from a requested peer, it assumes that the other peer is behind a firewall. Therefore it sends a Gnutella Push message to this host, to request a pushed download. The receiver of a Push message must immediately respond with a HTTP GIV message, containing the `File Index` and the `Servent Identifier` of the Push message. The file index is used to request the assigned `File Name` from the File manager. The `Servent Identifier` is a globally unique identifier representing the requesting host. Peers receiving a HTTP GIV message respond with a download request (HTTP GET message). The syntax of the GIV message is shown in Figure D.3.
HTTP GIV message can only be sent to Gnutella hosts, because only Gnutella hosts send Gnutella Push messages. Currently, firewalled hosts are not supported by P-Grid, but the problem will not be solved by an equivalent to the Gnutella Push message. Gridella only supports this message to be compatible with Gnutella.

Figure D.4 shows an example HTTP GIV message. The Gnutella Push message is sent by the requesting host of the previous example (Figure D.2) if the storing host is behind a firewall. The storing host uses the file index `458` and

```
GIV <File Index>:<Servent Identifier>/<File Name>\n\n
```

Figure D.3: HTTP GIV message syntax

the servent identifier `432DE235CA3EF523490F32DE6AC462EB` of the received Push message and appends the message with the file name `Tori.mp3`. The requesting host will respond to this HTTP GIV message with the HTTP GET message shown in Figure D.2.

```
GIV 458:432DE235CA3EF523490F32DE6AC462EB/Tori.mp3\n\n
```

Figure D.4: HTTP GIV message example

D.3 HTTP OK Message

A HTTP OK message is sent in response to a received HTTP GET message and is followed by the requested file. The message contains a response `Code` and a `Description`. The `Code` can be '200' with a `Description` 'OK' to indicate a complete download, or `Code` '206' and `Description` 'Partial Content' for partial downloads of files. The content length is computed by '`Range End` - `Range Begin` + 1' and indicates the length of the content following. Content range indicates the range in the file of the content following (represented by `Range Begin` and `Range End`), followed by the full length of the file (`Content Length`). The syntax of the message is shown in Figure D.5.

```
HTTP/1.1 <Code> <Description>\r\n
Server: Gridella\r\n
Content-length: (<Range End>-<Range Begin>+1)\r\n
Content-Range: bytes <Range Begin>-<Range End>/<Content Length>\r\n
\r\n
```

Figure D.5: HTTP OK message syntax

Figure D.6 shows an example HTTP OK message. It is the response message to the HTTP GET message in Figure D.2. As only a part of the file is requested the response code is `206`. The range was given in the HTTP GET message (`4678 - 12487`). Then the calculated content length is `7810`. At last the content length of the file is given (`3427643`).

```
HTTP/1.1 206 Partial Content\r\n
Server: Gridella\r\n
Content-length: 7810\r\n
Content-Range: bytes 4678-12487/3427643\r\n
\r\n
```

Figure D.6: HTTP OK message example

Bibliography

[1] Karl Aberer. P-Grid: A self-organizing access structure for P2P information systems. In *Proceedings of the Sixth International Conference on Cooperative Information Systems (CoopIS 2001)*, 2001. http://lsirwww.epfl.ch/publications/CoopIS2001.pdf.

[2] Karl Aberer. Scalable Data Access in P2P Systems Using Unbalanced Search Trees. In *Workshop on Distributed Data and Structures (WDAS-2002)*, 2002. http://www.p-grid.org/Papers/WDAS2002.pdf.

[3] Karl Aberer and Zoran Despotovic. Managing Trust in a Peer-2-Peer Information System. Technical Report DSC/2001/029, Swiss Federal Institute of Technology, Lausanne (EPFL), 2001. http:/lsirwww.epfl.ch/publications/P2P-Trust.pdf.

[4] Karl Aberer and Manfred Hauswirth. Semantic Gossiping. In *Database and Information Systems Research for Semantic Web and Enterprises, Invitational Workshop*, 2002. http://lsirpeople.epfl.ch/hauswirth/papers/SemWeb.pdf.

[5] Karl Aberer, Manfred Hauswirth, Magdalena Punceva, and Roman Schmidt. Improving Data Access in P2P Systems. *IEEE Internet Computing*, 6(1), January/February 2002. http://www.p-grid.org/Papers/IC2002.pdf.

[6] Eytan Adar and Bernardo A. Huberman. Free Riding on Gnutella. Technical report, Xerox PARC, 9 September 2000. http://www.firstmonday.dk/issues/issue5_10/adar/index.html.

[7] Audiogalaxy homepage, 2001. http://www.audiogalaxy.com/.

[8] Antonio Carzaniga, D.S. Rosenblum, and A.L. Wolf. Achieving Expressiveness and Scalability in an Internet-Scale Event Notification Service. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing*, July 2000. http://www.pdos.lcs.mit.edu/papers/chord:hotos01/hotos8.pdf.

[9] *The Gnutella Protocol Specification v0.4 (Document Revision 1.2)*, June 15 2001. http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf.

[10] Frank Dabek, Emma Brunskill, M. Frans Kaashoek, David Karger, Robert Morris, Ion Stoica, and Hari Balakrishnan. Building Peer-to-Peer Systems with Chord, a Distributed Lookup Service. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, 2001. http://www.pdos.lcs.mit.edu/papers/chord:hotos01/hotos8.pdf.

[11] FastTrack homepage, 2001. http://www.fasttrack.nu/.

[12] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. Network Working Group, January 1997. RFC 2068. http://www.ietf.org/rfc/rfc2068.txt.

[13] Freenet Project, 2001. http://FreenetProject.org.

[14] Gnutella homepage, 2001. http://gnutella.wego.com/.

[15] Li Gong. JXTA: A Network Programming Environment. *IEEE Internet Computing*, 5(3):88–95, May/June 2001. http://dlib.computer.org/ic/books/ic2001/pdf/w3088.pdf.

[16] Manfred Hauswirth, Anwitaman Datta, and Karl Aberer. Handling Identity in Peer-to-Peer Systems. Technical report, Laboratoire de Systèmes d'Information Répartis (LSIR), École Polytechnique Fédérale de Lausanne (EPFL), 2002. http://lsirpeople.epfl.ch/hauswirth/papers/TR-IC-2002-67.pdf.

[17] Dennis Heimbigner. Adapting Publish/Subscribe Middleware to Achieve Gnutella-like Functionality. In *2001 ACM Symposium on Applied Computing (SAC 2001): Special Track on Coordination Models, Languages and Applications*, March 2001. http://www.cs.colorado.edu/serl/papers/CU-CS-909-00.pdf.

[18] J. P. Hubaux, Th.Gross, J.-Y. Le Boudec, and M. Vetterli. Towards self-organized mobile ad-hoc networks: the Terminodes project. *IEEE Communications Magazine*, January 2001. http://www.terminodes.org/publications/commag01a.pdf.

[19] iMesh homepage, 2001. http://www.iMesh.com/.

[20] T. Imielinski, S. Viswanathan, and B. R. Badrinath. Data on Air: Organization and Access. *IEEE Transactions on Knowledge and Data Engineering*, 9(3), May/June 1997.

[21] Mihajlo A. Jovanovic, Fred S. Annexstein, and Kenneth A. Berman. Scalability Issues in Large Peer-to-Peer Networks - A Case Study of Gnutella. University of Cincinnati, Laboratory for Networks and Applied Graph Theory, 2001. http://www.ececs.uc.edu/~mjovanov/Research/paper.ps.

[22] Jon Kleinberg. The Small-World Phenomenon: An Algorithmic Perspective. Technical Report 99-1776, Cornell Computer Science, October 1999. http://www.cs.cornell.edu/home/kleinber/swn.pdf.

[23] Mojo Nation Technology Overview, 14 February 2000. http://www.mojonation.net/docs/technical_overview.shtml.

[24] Napster homepage, 2001. http://www.napster.com/.

[25] C. Greg Plaxton, Rajmohan Rajaraman, and Andréa W. Richa. Accessing Nearby Copies of Replicated Objects in a Distributed Environment. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 311–20, June 1997. http://www.cs.utexas.edu/users/plaxton/ps/1997/spaa.ps.

[26] Sean Rhea. Maintenance-free Global Data Storage. *IEEE Internet Computing*, 5(5), September/October 2001.

[27] Kunwadee Sripanidkulchai. The popularity of Gnutella queries and its implications on scalability. http://www.cs.cmu.edu/~kunwadee/research/p2p/gnutella.html, February 2001.

[28] Michael Stonebraker, Paul M. Aoki, Witold Litwin, Avi Pfeffer, Adam Sah, Jeff Sidell, Carl Staelin, and Andrew Yu. Mariposa: A Wide-Area Distributed Database System. *VLDB Journal*, 5(1):48–63, 1996. http://epoch.CS.Berkeley.EDU:8000/personal/aoki/papers/vldbj96.pdf.

[29] Kelly Truelove and Andrew Chasin. Morpheus Out of the Underworld, 2001. http://www.openp2p.com/pub/a/p2p/2001/07/02/morpheus.html.

[30] William Wong. Furi hompage. http://www.jps.net/williamw/furi/, 2001.

[31] Haruo Yokota, Yasuhiko Kanemasa, and Jun Miyazaki. Fat-Btree: An Update-Conscious Parallel Directory Structure. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 1999.