Research Report No. 2007:02

Peer–to–peer Traffic Measurements

Dragos Ilie
David Erman

Department of Telecommunication Systems,
School of Engineering,
Blekinge Institute of Technology,
S–371 79 Karlskrona, Sweden

© 2007 by Dragos Ilie and David Erman. All rights reserved.

Blekinge Institute of Technology
Research Report No. 2007:02
ISSN 1103-1581
Published 2007.
Printed by Kaserntryckeriet AB.
Karlskrona 2007, Sweden.

This publication was typeset using LaTeX.

Abstract

The global Internet has emerged to become an integral part of everyday life. Internet is now as fundamental a part of the infrastructure as is the telephone system or the road network. Peer-to-Peer (P2P) is the logical antithesis of the Client-Server (CS) paradigm that has been the ostensible predominant paradigm for IP-based networks since their inception.

Current research indicates that P2P applications are responsible for a substantial part of the Internet traffic. New P2P services are developed and released at a high pace. The number of users embracing new P2P technology is also increasing fast. It is therefore important to understand the impact of the new P2P services on the existing Internet infrastructure and on legacy applications. This report describes a measurement infrastructure geared towards P2P network traffic collection and analysis, and presents measurement results for two P2P applications: Gnutella and BitTorrent.

Contents

1 Introduction **1**
1.1 Motivation 2

2 Peer–to–Peer Protocols **3**
2.1 P2P Evolution 3
2.2 P2P Definitions 4
2.3 Distributed Hash Tables 5
2.4 P2P and Ad-Hoc Networks 6
2.5 P2P and File Sharing 7
2.6 P2P and the Grid 8
2.7 P2P and Sensor Networks 9

3 Protocol Descriptions **11**
3.1 BitTorrent 11
3.2 Gnutella 20

4 Measurement Software Description **29**
4.1 Passive measurements 29
4.2 Network infrastructure 31
4.3 TCP Reassembly Framework 32
4.4 Application Logging 35
4.5 Log Formats 36

5 BitTorrent **41**
5.1 Measurement details 41
5.2 Aggregate results 42
5.3 Swarm size dynamicity 45

6 Gnutella **47**
6.1 Session Statistics 47
6.2 Message Statistics 48
6.3 Transfer Rate Statistics 51

A BitTorrent Application Log DTD **55**

B Acronyms **57**

Bibliography **59**

List of Figures

3.1 BitTorrent handshake procedure. 14
3.2 Example announce GET request. 15
3.3 Example scrape GET request. 17
3.4 BitTorrent handshake procedure. 18
3.5 Example of a Gnutella session. 26

4.1 Measurement network infrastructures. 32
4.2 Measurement procedures. 33
4.3 Extract from BitTorrent XML log file. 38
4.4 Sample BitTorrent log file. 40

5.1 Temporal structure of measurements. 42
5.2 Swarm size for measurement 6. 45
5.3 Swarm size for measurement 10. 46

6.1 Gnutella Transfer Rates. 54
6.2 Gnutella Transfer Rates at IP layer. 54

List of Tables

2.1 P2P and CS content models. 5

5.1 Measurement summary. 42
5.2 Content summary. 43
5.3 Download time and average download rate summary. 43
5.4 Session and peer summary. 44
5.5 Downstream protocol message summary. 44
5.6 Upstream protocol message summary. 45

6.1 Incoming session statistics. 48
6.2 Outgoing session statistics. 48
6.3 Incoming + Outgoing session statistics. 48
6.4 Message size statistics. 49
6.5 Message duration statistics. 49
6.6 Message interarrival time statistics. 50
6.7 Message interdeparture time statistics. 50
6.8 Handshake message rate statistics. 51
6.9 Handshake byte rate statistics. 51
6.10 PING–PONG message rate statistics. 51
6.11 PING–PONG byte rate statistics. 51
6.12 QUERY–QUERY_HIT message rate statistics. 51
6.13 QUERY–QUERY_HIT byte rate statistics. 52
6.14 QRP and HSEP message rate statistics. 52
6.15 QRP and HSEP byte rate statistics. 52
6.16 PUSH and BYE message rate statistics. 52
6.17 PUSH and BYE byte rate statistics. 52
6.18 VENDOR and UNKNOWN message rate statistics. 53
6.19 VENDOR and UNKNOWN byte rate statistics. 53
6.20 Gnutella (all type) message rate statistics. 53
6.21 Gnutella (all type) byte rate statistics. 53
6.22 IP Byte rate statistics. 53

Chapter 1

Introduction

The global Internet has emerged to become an integral part of everyday life. Internet is now as fundamental a part of the infrastructure as is the telephone system or the road network. The driving factor pushing the acceptance and widespread usage of the Internet was the introduction of the World-Wide Web (WWW) by Tim Berners-Lee in 1989. The WWW provided ways of accessing information at rates and amounts then unimagined, and quickly became the Internet "killer application".

In May 1999, ten years after the advent of the WWW, Shawn Fanning introduces Napster, arguably the first modern P2P application. The Napster application and protocols were the first to allow users to share files among each other without the need of a central storage server. Very quickly, Napster became immensely popular, and the P2P revolution begun.

Since the advent of Napster, P2P systems have become wide-spread with the emergence of file-sharing applications such as Gnutella, Kazaa and eDonkey. These systems have generated headlines across the globe when the U.S. Recording Industry Association (RIAA http://www.riaa.com) and Motion Picture Association of America (MPAA http://www.mpaa.org) have filed law suits against file-sharing users suspected of copyright infringement. The law suits are partly responsible for the embrace of the term P2P as an equivalent for illegal file-sharing. Fortunately, the concept of P2P networking is broader than that and P2P systems have many useful and legal applications.

The P2P paradigm is the logical antithesis of the CS paradigm that has been the ostensible predominant paradigm for IP-based networks since their inception. This is however only true to a certain degree, as the idea of sharing among equals has been imbued in the Internet since the early days of the network. Two examples supporting this statement are the e-mail system employed in the Internet and the Domain Name System (DNS). Both protocols are so tightly connected to the inner workings of the Internet, that it is impossible to imagine the degree of usage that the Internet sees today in the absence of those protocols. Once an e-mail has left the user's mail software, it is routed among mail transfer agents (MTAs), all acting as as equally valued message forwarders. The DNS is the first distributed information database, and implements a hierarchical mapping scheme, which is comparable to the multi-layered P2P systems that have appeared the last few years, such as Gnutella.

The major difference between legacy P2P systems such as DNS and e-mail and the new systems such as Gnutella, Napster and eDonkey is that the older systems work as part of the network core, while the new applications are typically application-layer protocols run by edge-node applications. This shift of the edge nodes from acting as service users to being both service providers and users is significantly changing the characteristics of the network traffic.

This report describes a measurement infrastructure geared towards P2P network traffic collection and analysis. The current chapter has provided a brief introduction, which will be followed by the motivation for the work described in the report. Chapter two continues with a discussion of P2P networks and related network types. Chapter three provides a fairly detailed descriptions of the two protocols measured for the report, BitTorrent and Gnutella. Chapter four explains briefly various network measurement techniques and then presents the measurement infrastructure and software developed at Blekinge Institute of Technology (BTH). Chapter five and six present some of the results obtained from the network traffic measurements of BitTorrent and Gnutella protocols, respectively.

1.1 Motivation

The research group at the Department of Telecomunication Systems at BTH has traditionally been involved in research areas related to Internet traffic measurements.

Lately, P2P applications seem to be responsible for the predominant part of the Internet traffic [1]. New P2P services are developed and released at a high pace. The number of users embracing new P2P technology is also increasing fast. It is therefore important to understand the impact of the new P2P services on the existing Internet infrastructure and on legacy applications. It was natural in this context for the research focus at BTH to turn towards P2P traffic. This can be seen as part of a broader effort to adapt new services to coexist with older ones as well as to optimize the overall online user experience.

We applied several constraints on selecting which P2P protocols to study: open protocol specifications, open-source software implementation, large user base and access to discussion groups where system developers can be contacted. Open protocol specifications were essential because this means no reverse engineering or other type of guess-work is necessary to decode protocol messages, thus allowing the research to focus on measurements and analysis. Access to source code enabled us to understand how certain protocol features, not covered by specifications, were implemented by various vendors. A large user base guaranteed enough peers would be available for measurements. Access to system developers in discussion groups helped us sort out some of the more intricate details of specifications and implementation. Both Gnutella and BitTorrent satisfied all these constraints.

The P2P research at BTH has two goals. The first is to establish analytical traffic models that will offer new insights about the properties of P2P traffic. This will hopefully lead to more scalable P2P networks and more efficient and fair resource utilization. The second goal is to research new P2P services that can be built on top of the existing P2P infrastructure, leveraging the advantages offered by P2P systems (e.g., efficient data dissemination and load balancing).

In particular, our group is planning to perform research on issues related to routing in overlay networks: Quality of Service (QoS) guarantees, reliability, scalability and multipath routing and transport [2]. Furthermore, we would like to investigate the applicability of our results in ad-hoc, grid and sensor networks.

Chapter 2

Peer–to–Peer Protocols

The concept of P2P protocols in relation to data communications is quite broad. Generally, this means that nodes engaged in mutual data exchange are capable of equivalent functionality. This is in contrast to pure CS protocols, where nodes may either serve or be served data. A more formal definition of P2P protocols is provided in Section 2.2.

2.1 P2P Evolution

The earliest recorded use of the term "peer-to-peer" occured in 1984 and was the context of IBM's Advanced Peer to Peer Networking (APPN) architecture [3]. This was the result of multiple enhancements to the Systems Network Architecture (SNA).

Although early networking protocols such as NNTP and SMTP were working in a P2P fashion – indeed, the original ARPANET was designed as a P2P system – the term P2P did not become mainstream before the appearance of Napster in the fall of 1999.

Napster was the first P2P service with the goal to provide users with easy means of finding music files (MP3s). The architecture of Napster was built around a central server that was used to index music files shared by client nodes. This approach is called a *centralized directory*. The centralized directory allowed Napster to give a very rapid answer as to which hosts stored a particular file. The actual file transfer occured directly between the host looking for the file and the host storing the file.

The success of Napster became quickly a source of serious concern for major record companies, who rapidly filed a lawsuit against Napster on grounds of copyright infringement. The lawsuit made Napster immensely popular, attracting millions of additional users. However, Napster couldn't withstand the pressure of the lawsuit and in July 2001 they were forced to shut down the central server. Without the central server the client nodes could no longer search for files. Thus, the fragility of a centralized directory system became clear.

Napster is one of the first generation P2P applications as defined by [3]. Following the advent of Napster, several other P2P applications emerged. Similar in appearance, but altogether different beasts in detail. Gnutella [4], which was released by Justin Frankel of Winamp fame in early 2000, opted to implement a fully distributed system, with no central authority. The same year saw the emergence of the FreeNet system. FreeNet was the the brainchild of Ian Clarke, who had written his Master's thesis on a distributed, anonymous and decentralized information storage and retrieval system. This system later became FreeNet. FreeNet's major difference to previous P2P

systems was the complete anonymity that it offered users.

The fully distributed architecture was resilient to node failures and was also immune to service disruptions of the type experienced by Napster. However, experience with Gnutella showed that fully distributed P2P systems may lead to scalability problems due to massive amounts of signaling traffic [5].

By late 2000 and early 2001, the P2P boom had started in earnest, and applications such as KaZaA, DirectConnect, SoulSeek and eDonkey started appearing. These systems usually provided some form of community-like features such as chat rooms and forums, in addition to the file-sharing services provided by previous systems.

KaZaA, which uses the FastTrack protocol, introduced the concept of *supernodes* in order to solve scalability problems similar to those experienced by Gnutella. Each supernode manages a number of regular nodes and exchanges information about them with other supernodes. Regular nodes upload file lists and search requests to their supernode. The search requests are processed solely among supernodes. Regular peers establish direct HyperText Transfer Protocol (HTTP) connections in order to download files.

Gnutella resolved the scalability problem in a similar way. In Gnutella, supernodes are called *ultrapeers*.

The last few years have seen the evolution of the old systems to better utilize network resources, and also the emergence of new systems with the specific focus on efficient bandwidth utilization. The prime example of these is the BitTorrent system. Also, new systems tend to focus on using *Distributed Hash Tables (DHTs)*. DHTs force network topology and data storage to follow specific mathematical structures in order to optimize various parameters (e.g., minimize delay or number of hops). They are seen as a promising alternatives to the flooding algorithms required by routing in unstructured P2P networks.

2.2 P2P Definitions

There is no clear concensus regarding an exact definition of a P2P system. Schollmeier makes an attempt in [6] to define a P2P network. In general, the notion of a P2P network appears to be leaning towards some form of utlization of edge node resources by other edge node resources. The resource in question is commonly accepted to be files, and much research is being done on efficient localisation and placement of files. There also seems to be some concensus regarding the idea of *pure* and *hybrid* systems.

A P2P network is defined in [6] as a network in which the service offered by the system is provided by the participating nodes share part of some local resource pool, such as disk space, files, CPU processing time etc. A *pure* P2P network is one in which any given participant may be removed without the system experiencing loss of service. Examples of this type of networks are Gnutella, FastTrack and FreeNet. A *hybrid* P2P network is one in which a central authority of some sort is necessary for the system to function properly. Note that, in contrast to the CS model, the central authority in a hybrid network rarely share resources – this functionality is still provided by the participating peers. The central authority is commonly an indexing server for files or provides a peer localisation service. Examples of this type of network are Napster, eDonkey and DirectConnect.

It is also possible to take a resource view on the two types of P2P networks described above. We consider the functions of content insertion, distribution and control and how they are performed in P2P and CS networks. We summarize these in table 2.1.

Insertion Insertion is the function of adding content to the resource pool of a network. We refer here to insertion in the sense of providing the content, so that in both pure and hybrid networks content is inserted by the participating peers. This is analogous with the peers sharing content. In a CS system however, content is always provided by the server, and thus also “shared” by the server.

Distribution Distribution is the function of retrieving content from the resource pool of a network. Again, P2P systems lack central content localization, thus content is disseminated in a distributed fashion. This does not necessarily mean that parts of the same content is retrieved from different sources, i.e. swarming, but rather that the parts, e.g. files, of the total resource pool are retrieved from different sources.

By hybrid CS systems, we here refer to Content Delivery Networks (CDNs), such as Akamai [7], and redundant server systems in which several servers provide the same content, but are accessed by client from a single Universal Resource Locator (URL). This is a very common model for Web servers in the Internet today.

Control Control is the function of managing the resource pool of a network, such as admission control, resource localization etc. This is the primary function that separates the two types of P2P networks. The peers participating in fully distributed networks are required to assist in the control mechanisms in the network, while the hybrid systems may rely on a central authority for this. Of course, the clients in CS systems have no responsibility towards the network control functionality.

Table 2.1: *P2P and CS content models.*

	Pure P2P	Hybrid P2P	Hybrid CS	Pure CS
Insertion	Distributed	Distributed	Central	Central
Distribution	Distributed	Distributed	Central/Distributed	Central
Control	Distributed	Central	Central	Central

In addition to the definitions above, [3] also classifies P2P systems according to their “generation”. In this classification scheme, hybrid systems such as Napster are considered first generation systems, while fully decentralized systems such as FastTrack and Gnutella are second generation systems. A third generation is discussed as being the improvement upon the first two with respect to features such as redundancy, reliability or anonymity.

2.3 Distributed Hash Tables

Hash tables are data structures used for quick lookups of information (data).

A hash record is defined as a *key* with the corresponding *value*. Hash keys are typically numerical values or strings, while the hash values are indexes in an array and therefore are usually numerical. The array itself is called the *hash table*. A *hash function* operates on a given hash key producing a corresponding unique hash value (index) that points to a location in the the hash table. This is the location where the data is stored or the best place to start a search for it. The quality of a hash table and its hash function is related to the probability of *collisions*. A collision happens when two

or more keys point to the same location in the hash table. This problem can be solved by enlarging and rearranging the hash table, but will in general lead to severe performance degradation.

DHTs are hash tables spread across many nodes in a network. Each node participating in a DHT is responsible for a subset of the DHT keys. When the hash function is given a key it will produce a hash value that identifies the node responsible for that particular key.

In general, DHTs require the use of a *structured* overlay network. This means that the network topology must follow a particular mathematical (geometric) structure (e.g., a circle as in Chord [8] or a hypercube as in CAN [9]). The structure is typically mapped on some interval equivalent to $\Re$ or $\Re^n$. Each key gets a well defined position (coordinates) in the specific space interval. A host that joins the DHT is allocated a position in the same space interval. The host will then "own" the coordinates in some neighbourhood of its position. This is how keys are distributed to hosts. Search algorithms can then exploit the properties of the structure to quickly locate which host is owning what key. This is called DHT-routing or key based routing. For example, CAN uses a d-dimensional Cartesian coordinate space and its routing exploits this property by choosing the smallest Cartesian distance to the destination. A good comparison of the tradeoffs between different DHT geometries is presented in [10].

Although DHTs appear to be a very efficient form of information dissemination, storage and lookup, they also suffer from a number of problems [11]:

Scalability: The cost to maintain the integrity of the DHT when nodes join/leave the overlay is rather high.

Load balancing: Some keys are much more popular than others. Hosts responsible for these keys must handle a disproportionally high volume of traffic compared to other nodes. This phenomena is refered to as "hot spots".

Search flexibility: DHTs cannot handle efficiently keyword searches, at least not as efficient at exact searches.

Heterogeneity adaptation: In a heterogeneous environment nodes have different storage, memory, transmission and processing capabilities. Since the topology in a DHT must adapt to a specific structure, it usually does not have enough flexibility to take the heterogeneity factors into account.

Current research is addressing these problems [12].

2.4 P2P and Ad-Hoc Networks

During the late 1990s cheap wireless network cards for portable computers became widely available, with 802.11 emerging as the *de-facto* standard. They were used mostly in office-like environments together with one or more wireless base stations, also called Access Points (APs). The base stations offer services such as frequency allocation, authentication and authorization. The maximum distance from the AP, where radio communication with it is still possible, defines the (radio) cell radius.

In the absence of repeaters or high-performance antennas, the wireless cards have limited range on the order of a few hundred meters. In order to communicate with wireless units that are outside the range, they forward the data to the nearest AP which, using either the wired or wireless network, would further forward the data to the destination.

The type of wireless network described above is called a Wireless LAN (WLAN) in *infrastructure mode*. When only one AP is used, the units movement is confined to the cell radius. This setup is called Basic Structure Service (BSS). Sometimes it is desirable to cover a larger area, such as a whole building or campus. This can be done by using several APs that perform handover when wireless units move between them. In this case the set of participating APs and wireless units is called Extended Service Set (ESS). The ESS mode hides implementation details (e.g., communication between base stations) from the network layer presenting it with what appears to be a regular link layer. Thus, the handovers are transparent for the network layer [13].

Infrastructure mode may not be an option for certain type of environments such as geographically isolated sites, disaster areas or dynamic battlefields. In this type of scenario, the wireless units must be able to communicate with each other in the absence of base stations. Two units that are outside radio range from each other and wish to exchange data will use intermediate units (within radio range) to route the data to destination. This type of communication mode is called *ad-hoc mode* and the network is called a mobile ad-hoc network (MANET).

In essence, each wireless host in the ad-hoc WLAN acts as a router for the other hosts. There is currently a large variety of ad-hoc routing protocols. A complete description is outside the scope of this report, but the interested reader may start with [14]. The important thing to this discussion is that each wireless unit supplies and demands the same type of services from the other WLAN units. This is a typical form of P2P computing according to the P2P definition in [6].

2.5 P2P and File Sharing

File sharing is almost as old as operating systems themselves. Early solutions include protocols such as UNIX remote copy (rcp) command and the File Transfer Protocol (FTP). They were quickly followed by full-fledged network file systems such as NFS and SAMBA. Common for these protocols (with the exception of rcp) is that they were shaped around the CS paradigm, with the servers being the entity storing and serving files. A client that wants to share files must upload them to a server to make them available to other clients.

Instant messaging systems such as ICQ[15], Yahoo! Messenger[16] and Microsoft Messenger[17] attempted to solve this problem by implementing a mechanism similar to rcp. Users could thus share file with each other without having to store them on a central server. In fact, this was the first form of P2P filesharing. Napster further extended this idea by implementing efficient file search facilities.

In the public eye, P2P is synonymous with file sharing. While other applications that may be termed P2P, e.g. the SETI@home project [18], distributed.net [19] and ZetaGrid [20] have been rather successful in attracting a user-base, no other type of service come close to attracting the number of users that filesharing services have. Services such as those mentioned here are examples of altruistic systems in the sense that the participating peers provide CPU processing power and time to a common resource pool that is then used to perform various complex calculations such as calculating fast Fourier transforms of galactical radio data, code-breaking or finding roots of the Riemann Zeta-function.

One of the reasons for the difference in number of users could be that the incentive to altruistically share resources without gaining anything other than some virtual fame or feel-good points of having contributed to the greater good of humanity seems to be low. Most file sharing P2P systems employ some form of admission scheme in which peers are not allowed to join the system or download from it unless they are sharing an adequate amount of files. This provides a dual incentive: first, a peer wanting to join the network must[1] provide sort of an entry token in the form of shared

[1] Not in all systems, but in most hybrid systems.

files, and second, peers joining the system know that there is a certain amount of content provided to them once they join. The BitTorrent P2P system is one of the most prominent networks in enforcing incentive, though admission control in a BitTorrent network is typically handled through web-based registration systems.

As not all files are equally desirable in every system, files not belonging to the general category of files handled in a specific P2P network should not be allowed in. For instance, a network such as Napster, which only managed digital music files, might not be interested in peers sharing text files. For systems that require a large amount of file data to be shared as an admission scheme, this becomes a problem. Peers may share “junk files” just to gain access to the network. Junk files are files that are not really requested or desired in the network. This practice is usually scorned upon, but is hard to get to grips with. Some systems, such as eDonkey have implemented a rating system, in which peers are punished for sharing junk files.

Similar to junk files, there are also “fakes” or “decoys”. Fakes are files inserted in the network that masquerade under a filename that does not represent the actual content, or files that contain modified versions of the same content. By adding fakes into the network, the real content is made more difficult to find. This problem is alleviated by using various hashing techniques for the files instead of only relying on the filenames to identify the content. An example of this is the insertion of a faked Madonna single, in which the artist had overlaid a phrase on top of her newly released single.

While file sharing in and of itself is not an illegal technology and has several non-copyright infringing uses, the ease with which peers may share copyrighted material has drawn the attention of the MPAA (Motion Picture Association of America) and RIAA (Recording Industry Association of America). These organizations are of the view that sharing of material under the copyrights of their members is seriously harming their revenue streams, by decreasing sales. In 2004, the MPAA and RIAA started suing individuals for sharing copyrighted material. However, not all copyright holders and artists agree on this course of action, nor do they agree on the detrimental effect file sharing has on sales or artistic expression. Several smaller record labels have embraced the distribution of samples of their artist’s music online, and artists have formed coalitions against what they feel is the oppressive behaviour of the larger record labels.

More recently, P2P systems have been employed by corporations to distribute large files such as Linux distributions, game demos and patches. Many companies make use of the BitTorrent system for this, as it provides for substantial savings in bandwidth costs.

2.6 P2P and the Grid

The goal of distributed computing is to provide a cost-efficient alternative to expensive supercomputers by harnessing the collective processing power of a group of general purpose workstations. This approach lead to computer clusters such as Beowulf [21]. Computer clusters typically involve tens or hundreds of computers owned by a company or organization, which are interconnected in a LAN configuration.

With the growing popularity of Internet, scientists and engineers have been looking at ways of distributing computations to Internet-connected computers with free computing cycles to share. This approach typically involves several hundred thousand computers spread across the world and is referred to as *Internet computing*. It differs from clusters in two ways: the number of hosts participating and the geographic spread.

Internet computing is part of greater effort called *Grid computing*. Grid computing takes on a holistic view focusing not only on distributing computations but also catering for resource allo-

cation, job scheduling, security and collaboration between users and organizations. It does that by providing open architecture and protocol specifications and by defining services to be provided by grid members. Examples of Grid computing efforts include the Globus Toolkit [22, 23] and BOINC [24].

P2P is related to Grid and Internet computing through the direct exchange of resources and services that takes place between peers. The above mentioned ZetaGrid and SETI@home could be viewed as a type of "proto-grid"[2], and the distributed screensaver ElectricSheep [25] is a more light-hearted variant of the "proto-grid" systems.

The main difference between the Grid and P2P is that "Grid computing addresses infrastructure but not yet failure, whereas P2P addresses failure but not yet infrastructure" [26]. For a good comparison of P2P and Grid computing see [26] and [27].

2.7 P2P and Sensor Networks

The term sensor network refers to a wireless communication network consisting of a rather large number (hundreds to thousands) of small-sized, densely deployed, electronic devices (sensors) that perform measurements in their immediate vicinity and transfer the results to a computation center in a hop-by-hop fashion [28]. The geographic area covered by a sensor network is called a *sensor field*.

Sensor networks are similar to ad-hoc networks with reference to data routing. Both type of networks require each node to actively participate in routing decisions (and data forwarding) in order to allow for communications across distances larger than the radio range of a single node. The main difference is that sensors have very stringent requirements for low power and low computation operation. The low power requirement generally means that nodes have smaller radio range than ad-hoc networks. This is the reason why they are densly deployed. The low computation requirement means that they typically use CPUs with limited capabilities in order to conserve energy (i.e., battery power). This means that they often employ special purpose routing protocols that require less computation.

A major difference between sensor networks and other type of networks (e.g., ad-hoc, grid and wired) is that sensor networks are *data-centric* as opposed to *node-centric*. For example, in a data-centric network the user or the application will ask which node (e.g., sensor) is exceeding a specific temperature. In contrast, in a node-centric network the user or application would rather ask what is the temperature of a specific node [29]. A data-centric approach will typically require attribute-based naming. An attribute is a name for a particular type of data. Lists of attributes define the type of data a node is interested in. For example the attribute list "`grid=50, temperature>30C`" asks for data from grid 50 from sensors that measure a temperature exceeding 30 degrees Celsius.

The data centric approach coupled with the low power, low computation operation requirements and the number of sensors in a sensor field makes it impractical to monitor each node at all times. Therefore, the focus is instead on self-organizing sensor networks that cluster nodes in a manner that facilitates local coordination in pursuit of global goals [29]. This lead to research into localized algorithms for coordination. Currently, *directed diffusion* appears to be a very strong candidate. In directed diffusion nodes establish gradients of interest for specific data. The gradients lead to the natural formation of routes between data providers (sources) and data consumers (sinks). Details on directed diffusion are found in [30].

In order to minimize the volume of data transfers, each node in a sensor network performs data

[2]The term proto-grid is used to denote a first generation grid that may have left out some of the requirements for a true grid implementation.

fusion (data aggregation). Data fusion works by combining data described by a specific attribute, which was received from different nodes. Data-centric routing may lead to problems related to implosion and overlap deficiencies [31]. The implosion problem is caused by multiple nodes sending the exact same data to their neighbours. The overlap problem is similar to implosion and appears when nodes covering different geographical areas send overlapping data. Data fusion attempts to solve both problems.

P2P networks and sensor networks share similarities with ad-hoc routing. Further, P2P networks are becoming more data centric, in particular when they use DHTs. For example, in a DHT-based file sharing application the users are interested in which nodes host a specific *file* rather then being interested in what files a specific *node* is hosting.

Chapter 3

Protocol Descriptions

3.1 BitTorrent

BitTorrent is a P2P protocol for content distribution and replication designed to quickly, efficiently and fairly replicate data [32, 33]. The BitTorrent system may be viewed as being comprised of two protocols and a set of resource metadata. The two protocols are used for communication among peers, and for the communication with a central network entity called the *tracker*. The metadata provides all the information needed for a peer to join in a BitTorrent distribution swarm and to verify correct reception of a resource.

We use the following terminology for the rest of the report: a *BitTorrent swarm* refers to all the network entities partaking in a distribution of a specific resource. When we refer to the *BitTorrent protocol* or *protocol* in singular, we refer to the peer–peer protocol, while explicitly referring to the *tracker protocol* for the peer–tracker communication. The collection of protocols (peer, tracker and metadata) are referred to as the *BitTorrent protocol suite* or *protocol suite*.

In contrast to many other P2P protocols such as eDonkey [34], DirectConnect [35], KaZaA [36], the BitTorrent protocol suite does not provide any resource query or lookup functionality. Nor does it provide any chat or messaging facilities. The protocols rather focus on fair and effective replication and distribution of data. The signaling is geared towards an efficient dissemination of data only.

Fairness in the BitTorrent system is implemented by enforcing *tit-for-tat* exchange of content between peers. Non-uploading peers are only allowed to download very small amounts of data, making the download of a complete resource very time consuming if a peer does not share downloaded parts of the resource.

With one exception, the protocols operate over Transport Control Protocol (TCP) and use swarming, i.e., peers simultaneously downloading parts, so-called *pieces*, of the content from several peers simultaneously. The rationale for this is that it is more efficient in terms of network load, as the load is shared across links between peers. This results in a more evenly distributed network utilization than conventional CS distribution systems such as, e.g., FTP or HTTP.

The size of the pieces is fixed on a per-resource basis and cannot be changed. The default piece size is 2^{18} bytes. The selection of an appropriate piece size is a fairly important issue. If the piece size is small, re-downloading a failed piece is fast, while the amount of extra data needed to describe all the data in the resource grows. Larger piece sizes means less metadata, but longer re-download times.

3.1.1 BitTorrent Encoding

BitTorrent uses a simple encoding scheme for most of its protocol messages and associated data. This encoding scheme is known as *bencoding*. The scheme allows for data structuring and type definition, and currently supports four data types: strings, integers, lists and dictionaries.

strings Strings are encoded length-prefixed. The length should be given in base ten, and ASCII coded. The length should be followed by a colon, immediately followed by the specified number of characters as string data.

Note that the string encoding does not nessecarily mean that the string data are humanly readable, i.e., in the printable ASCII range. Strings carry any valid 8-bit value, and are commonly used to carry binary data.

Example: `3:BTH` encodes the string "BTH".

integers Integers are encoded by enclosing a base ten ASCII coded numerical string by `i` and `e`. Negative numbers are accepted, but not leading zeroes, except in the case for the value 0 itself.

Example: `i23e` encodes the integer 23.

lists Lists are encoded by enclosing any valid bencoding type, including other lists, by `l` and `e`. More than one type is allowed.

Example: `l3:agei30ee` encodes the string "age" and the integer 30.

dictionaries Dictionaries are encoded by enclosing *(key, value)* pairs by `d` and `e`. The keys must be bencoded strings and the values may be any valid bencoding type, including other dictionaries.

Example: `d3:agei30e4:name5:james5likesl4:food5:drinkee` encodes the structure:

```
age: 30
name: james
likes: {food, drink}
```

3.1.2 Resource Metadata

A peer interested in downloading some content by using BitTorrent must first obtain a set of metadata, the so-called *torrent* file, to be able to join a set of peers engaging in the distribution of the specific content. The metadata needed to join a BitTorrent swarm consists of the network address information (in BitTorrent terminology called the *announce URL*) of the tracker and resource information such as file and piece size. The torrent file itself is a bencoded version of the associated meta information.

An important part of the resource information is a set of Secure Hash Algorithm One (SHA-1) [37, 38] hash values[1], each value corresponding to a specific piece of the resource. These hash values are used to verify the correct reception of a piece. When rejoining a swarm, the client must recalculate the hash for each downloaded piece. This is a very intensive operation with regards to both CPU usage and disk I/O, which has resulted in certain alternative BitTorrent clients storing information regarding which pieces have been successfully downloaded within a specific field in the torrent file.

A separate SHA-1 hash value, the *info* field, is also included in the metadata. This value is used as an identification of the current swarm, and the hash value appears in both the tracker

[1] These are also known as *message digests*.

and peer protocols. The value is obtained by hashing the entire metadata (sans the *info*-field itself). Of course, if a third-party client has added extra fields to the torrent file that may change intermittently, such as the resume data mentioned above, these should not be taken into account when calculating the *info*-field hash value.

The metadata as defined by the original BitTorrent design does not contain any information regarding the peers participating in a swarm, though this information is added by some alternative clients to lessen strain on trackers when rejoining a swarm.

3.1.3 Network Entities and Protocols

A BitTorrent swarm is composed of *peers* and at least one *tracker*. The peers are responsible for content distribution among each other. Peers locate other peers by communicating with the tracker, which keeps peer lists for each swarm. A swarm may continue to function even after the loss of the tracker, but no new peers are able to join.

To be functional, the swarm initially needs at least one connected peer to have the entire content. These peers are denominated as *seeds*, while peers that do not have the entire content, i.e., downloading peers, are denominated as *leechers*.

The BitTorrent protocols (except the metadata distribution protocol) are the *tracker protocol* and the *peer protocol*. The tracker protocol is either a HTTP-based protocol or a UDP-based compact protocol, while the peer protocol is a BitTorrent-specific binary protocol. Peer-to-tracker communication usually takes place using HTTP, with peers issuing HTTP GET requests and the tracker returning the results of the query in the returning HTTP response.

The purpose of the peer request to the tracker is to locate other peers in the distribution swarm and to allow the tracker to record simple statistics of the swarm. The peer sends a request containing information about itself and some basic statistics to the tracker, which responds with a randomly selected subset of all peers engaged in the swarm.

The Peer Protocol

The peer protocol, also known as the peer wire protocol, operates over TCP, and uses in-band signaling. Signaling and data transfer are done in the form of a continuous bi-directional stream of length-prefixed protocol messages over a common TCP byte stream.

A BitTorrent session is equivalent with a TCP session, and there are no protocol entities for tearing down a BitTorrent session beyond the TCP teardown itself. Connections between peers are single TCP sessions, carrying both data and signaling traffic.

Once a TCP connection between two peers is established, the initiating peer sends a handshake message containing the peer id and info field hash (Figure 3.1). If the receiving peer replies with the corresponding information, the BitTorrent session is considered to be opened and the peers start exchanging messages across the TCP streams. Otherwise, the TCP connection is closed. Immediately following the handshake procedure, each peer sends information about the pieces of the resource it possesses. This is done only once, and only by using the first message after the handshake. The information is sent in a *bitfield* message, consisting of a stream of bits, with each bit index corresponding to a piece index.

The BitTorrent peer wire protocol has the following protocol messages:

piece The only payload-related protocol message. The message contains one sub-

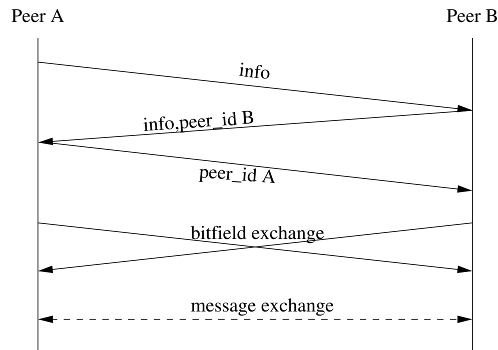


Figure 3.1: *BitTorrent handshake procedure.*

piece.

request The *request*-message is the method a peer wishing to download uses to notify the sending peer what *subpieces* is desired.

cancel If a peer has previously sent a *request* message, this message may be used to withdraw the request before it has been serviced. Mostly used during end-game mode[2].

interested This message is sent by a peer to another peer to notify it that the former intends to download some data. See Section 3.1.4 for a description of this and the following three messages.

not interested This is the negation of the previous message. It is sent when a peer no longer wants to download.

choke This message is send by a data transmitting peer to notify the receiving peer that it will no longer be allowed to download.

unchoke The negation of the previous message. Sent by a transmitting peer to a peer that has previously sent an *interested* message to the former.

have After a completed download, the peer sends this message to all its connected peers to notify them of which parts of the data are available from the peer.

bitfield Only sent during the initial BitTorrent handshake, and is then exchanged between the connecting peers. Contains a bitfield indicating which pieces the peer has.

keepalive Empty message, to keep a connection alive.

The Tracker Protocol

The tracker is accessed by HTTP or HTTPS GET requests. The default listening port is 6969. The tracker address, port and top-level directory are specified in the *announce_url* field in the torrent file for a specific swarm.

[2]End-game mode occurs when a peer only has very few pieces left to download. The peer requests these pieces from all connected peers, and downloads from whoever answers the quickest, and cancels the rest of the requests.

Tracker queries

Tracker queries are encoded as part of the GET URL, in which binary data such as the *info_hash* and *peer_id* are escaped as described in RFC1738 [39]. The query is added to the base URL by appending a questionmark, `?`, as described in RFC2396 [40].

The query itself is a sequence of `parameter=value` pairs, separated by ampersands, `&`, and possibly escaped. An example request is given in Figure 3.2.

```
GET /announce?info_hash=n%05hV%A9%BA%20%FC%29%12%1Ap%D4%12%5D%E6U%0A%85%E1&\
peer_id=M3-4-2--d0241ecc3a07&port=6881&key=0fcca260&uploaded=0&downloaded=0&\
left=663459840&compact=1&event=started HTTP/1.0
```

Figure 3.2: *Example announce GET request.*

Each announce request *must* include the following parameters:

info_hash The SHA-1 hash of the value contained in the *info* field in the torrent file.

peer_id A 20-byte string to uniquely identify the requesting peer. There is no concensus regarding the generation of this value, but several distinct types of ID-generation have appeared that may be used to identify which client a peer is running.

There is some disagreement between the official protocol description [41] and the Wiki [33]. The original specification states that this field most likely will have to be URL escaped, while the other claims that it *must not* be escaped.

port The listening port of the client. The default port range for the reference client is 6881–6889. Each active swarm needs a separate port in the default client, but third party clients have implemented single-port functionality.

uploaded The total number of bytes uploaded to all peers in the swarm, encoded in base ten ASCII. The specification does not state whether this takes into account re-transmits or not.

downloaded The total number of bytes downloaded from all peers in the swarm, encoded in base ten ASCII. The specification does not state whether this takes into account re-transmits or not.

left The total number of bytes left to download, also encoded in base ten ASCII.

The following parameters may optionally be included:

compact If set to 1, the tracker response will not be a proper bencoded datum as described below, but rather a binary list of peer addresses and ports. This list is encoded as a six-byte datum for each peer, in which the first six bytes are the IP address of the peer, and the last two bytes are the peer's listening port. This saves quite a bit of bandwidth, but is only usable in an IPv4 environment.

numwant Specifies the number of peers that the requesting peer is requesting from the tracker.

event May be one of:

- **started** The first request to the tracker, *must* include this parameter–value pair.

stopped If shutting down, this should be specified to indicate graceful shutdown.

completed Included to notify the tracker once a download is complete, and should not be included when joining a swarm with the full content.

key Used as session identifier.

Tracker replies

The tracker HTTP response, unless the *compact* parameter is 1, is a bencoded dictionary with the following fields:

interval Indicates the number of seconds between subsequent requests to the tracker.

complete Number of seeds in the swarm.

incomplete Number of leechers in the swarm.

peers Contains a list of dictionaries. Each dictionary in this list has the following keys:

- **peer id** The *peer_id* paramater that the peer has reported to the tracker.
- **ip** IP address or DNS name of the peer.
- **port** Listening port of the peer.

If the request fails for some reason, the dictionary only contains a *failure reason*-key, which contains a string indicating the reason for the failed request.

Tracker UDP protocol extension

To lower the bandwidth usage for heavily loaded trackers, a UDP-based tracker protocol has been proposed [42].

The UDP tracker protocol is not part of the official BitTorrent specification, but has been implemented in some of the third-party clients and trackers.

Compared to the standard HTTP-based protocol, the UDP protocol uses about 50 % less bandwidth. It also has the advantage of being stateless, as opposed to the stateful TCP connections required by the HTTP scheme, which means that a tracker is less likely to run out of resources due to things like half-open TCP-connections.

The scrape convention

Web scraping is the name for the procedure of parsing a web page to extract information from it. In BitTorrent, trackers are at liberty to implement functionality to allow peers to request information regarding a specific swarm without resorting to error-prone web-scraping techniques.

If the last name in the *announce url*, i.e. the name after the last `/`-character is `announce`, then the tracker supports scraping by using the *announce url* with the name `announce` replaced by `scrape`.

The scrape request may contain a *info_hash* parameter, as shown in Figure 3.3, or be completely without parameters.

```
GET /scrape?info_hash=n%05hV%A9%BA%20%FC)%12%1Ap%D4%12%5D%E6U%0A%85%E1 HTTP/1.0
```

Figure 3.3: *Example scrape GET request.*

The tracker will respond with a bencoded dictionary containing information about all files that the tracker is currently tracking. The dictionary has a single key, *files*, whose value is another dictionary whose keys are the 20-bit binary *info_hash* values of the torrents on the specific tracker. Each value of these keys contains another dictionary with the following fields:

complete — Number of seeds in the swarm.

downloaded — Number of registered `complete`-events for the swarm.

incomplete — Number of leechers in the swarm.

name — This optional field contains the name of the file as defined in the *name*-field in the torrent file.

3.1.4 Peer States

A peer maintains two states for each peer relationship. These states are known as the *interested* and *choked* states. The *interested* state is imposed by the requesting peer on the serving peer, while for the case of the *choked* state the opposite is true. If a peer is being choked, then it will not be sent any data by the serving peer until unchoking occurs. Thus, unchoking is usually equivalent with uploading.

The *interested* state indicates whether other peers have parts of the sought content. Interest should be expressed explicitly, as should lack of interest. That means that a peer wishing to download notifies the sending peer (where the sought data is) by sending an *interested* message, and as soon as the peer no longer needs any other data, a *not interested* message is issued. Similarly, for a peer to be allowed to download, it must have received an *unchoke* message from the sending peer. Once a peer receives a *choke* message, it will no longer be allowed to download. This allows the sending peer to keep track of the peers that are likely to immediately start downloading when unchoked. A new connection starts out choked and not interested, and a peer with all data, i.e., a seed, is never interested.

In addition to the two states described above, some clients add a third state – the *snubbed* state. A peer relationship enters this state when a peer purports that it is going to send a specific sub-piece, but fails to do so before a timeout occurs (typically 60 seconds). The local peer then considers itself snubbed by the non-cooperating peer, and will not consider sub-pieces requested from this peer to be requested at all.

3.1.5 Sharing Fairness and Bootstrapping

The *choke/unchoke* and *interested/not interested* mechanisms provides fairness in the BitTorrent protocol. As it is the transmitting peer that decides whether to allow a download or not, peers not sharing content will be reciprocated in the same manner.

To allow peers that have no content to join the swarm and start sharing, a mechanism called *optimistic unchoking* is employed. Optimistic unchoking means that from time to time, a peer with content will allow even a non-sharing peer to download. This will allow the peer to share the small portions of data received and thus enter into a data exchange with other peer.

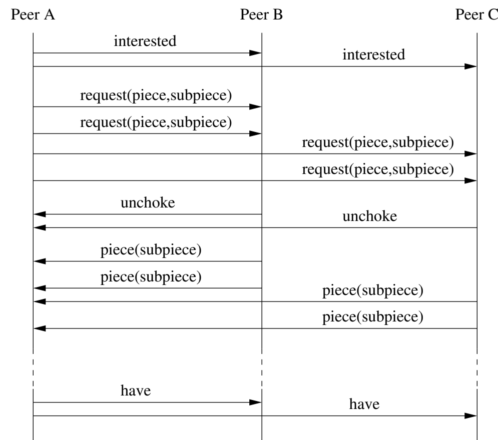


Figure 3.4: *BitTorrent handshake procedure.*

This means that while sharing resources is not strictly enforced it is strongly encouraged. It also means that peers that have not been able to configure their firewalls and/or Network Address Translation (NAT) routers properly will only be able to download the pieces altrustically shared by peers through the optimistic unchoking scheme.

3.1.6 Data Transfer

Data transfer is done in parts of a *piece* (called *sub-piece*, *block* or *chunk*) at a time, by issuing a *request* message. Sub-piece sizes are typically of size 16384 or 32768 bytes.

To allow TCP to increase throughput, several requests are usually sent back-to-back. Each request should result in the corresponding sub-piece to be transmitted. If the sub-piece is not received within a certain time (typically one minute), the non-transmitting peer is snubbed, i.e., it is punished by not being allowed to download, even if unchoked. Data transfer is done by sending a piece message, which contains the requested sub-piece (Figure 3.4). Once the entire piece, i.e., all sub-pieces, has been received, and the SHA-1 hash of the piece has been verified, a *have* message is sent to all connected peers.

The *have* message allows other peers in the swarm to update their internal information on which pieces are available from which peers.

End-game mode

When a peer is approaching completion of the download, it sends out requests for the remaining data to all currently connected peers to quickly finish the download. This is known as the *end-game mode*. Once a requested subpiece is received, the peer sends out `cancel`-messages to all peers that have not yet sent the requested data.

Without the end-game mode, there is a tendency for peers to download the final pieces from the same peer, which may be on a slow link [41].

3.1.7 BitTorrent Performance Issues

Even though BitTorrent has become very popular among home users, and widely deployed in corporate environments, there are still some issues currently being addressed for the next version of BitTorrent.

The most pressing issue is the load on the central tracker authority. There are two main problems related to the tracker: peak load and redundancy. Many trackers also handle more than a single swarm. The most popular trackers handle several hundred swarms simultaneously. It is not uncommon for popular swarms to contain hundreds or even thousands of peers. Each of these peers connect to the tracker every 30 minutes by default to request new peers and provide transfer statistics. An initial peer request to the tracker results in about 2-3 kB of response data. If these requests are evenly spread out temporally, the tracker can usually handle the load. However, if a particularly desired resource is made available, this may severely strain the tracker, as it will be subject to a mass accumulation of connections akin to a distributed denial of service attack by requesting peers. This is also known as the *flash-crowd effect* [43].

It is imperative for a swarm to have a functioning tracker if the swarm is to gain new peers since, without the tracker, new peers have no location to receive new peer addresses. Tracker redundancy is currently being explored and two alternatives are studied: backup trackers and distributing the tracking functionality in the swarm itself. An extension exists to the current protocol that adds a field, `announce-list`, to the metadata, which contains URLs to alternate trackers. No good way of distributing the tracking in the swarm has yet been found, but a network of distributed trackers has been proposed. Proposals of peers sending their currently connected peers to each other have also cropped up, but again, no consensus has been agreed on. Additionally, DHT functionality has been implemented in third party clients to address this problem [44]. A beta version of the reference client also has support for DHT functionality.

Another important problem is the initial sharing delay problem. If a torrent has large piece sizes, e.g., larger than 2 MB, the time before a peer has downloaded an entire piece and can start sharing the piece can be quite substantial. It would be preferable to have the ability to have varying verification granularities for the data in the swarm, so that a downloading peer does not have to wait for an entire piece to begin calculating the hashes of the data. One way to do this would be to use a mechanism known as Merkle trees [45], which allow for varying granularity. By using this mechanism, a peer may start sharing after having downloaded only a small amount of the data (on about the same order as the subpiece sizes).

3.1.8 Super Seeding

When a swarm is fairly new, i.e., there are few seeds in the swarm and peers have little of the shared resource, it makes sense to try to evenly distribute the pieces of the content to the downloading peers. This will speed up the dissemination of the entire content in the swarm. A normal seed would announce itself as having all pieces during the initial handshaking procedure, thus leaving the piece selection up to the downloading peer. Seeds have usually been in the swarm longer. This means that they are likely to have a better view on which pieces are the most rare in the swarm, and thus most suitable to be first inserted. As soon as peers start receiving the rare pieces, other peers can download them from other peers instead of seeds. This further balances the load in and increases the performance of the swarm.

A seed that employs *super seeding* does not advertise having any pieces at all during handshake. As peers connect to the in effect hidden seed, it instead sends `have`-messages on a per-peer basis to entice specific peers to download a particular piece.

This mechanism is most effective in new swarms, or when there is a high peer-to-seed ratio and the peers have little data. It is not recommended for everyday use.

As certain peers might have heuristics governing which swarms to be part of, a swarm containing *only* super seeds might be discarded. This is because peers cannot detect the super seed as a seed, thus assuming that the swarm is unseeded. This decreases the overall performance of the swarm.

3.2 Gnutella

Gnutella is a decentralized P2P system. Participants can share any type of resources, although the currently available specification covers only file resources. The first "official" Gnutella protocol was version 0.4 [4]. Soon, Gnutella version 0.6 [46] was released with improvements based on the lessons learned from version 0.4. The protocol is easily extendable, which has lead to a variety of proprietary and non-proprietary extensions (e.g., Ultrapeers and the Query Routing Protocol (QRP)). For a while, the two protocol versions lived side by side and improvements were merged from the v0.6 line into the legacy v0.4 line. However, it seems that July 1st 2003 was sort of a "flag day" when Gnutella v0.4 peers were blocked from the network[3].

The activities of Gnutella peers can be divided into two main categories: signaling and user data exchange (further referred to as data exchange).

The signaling activities are concerned with discovering the network topology and locating resources. Data exchange occurs when a peer has localized a resource of interest (e.g., a document file). The peer downloads files over direct HTTP connections.

3.2.1 Ultrapeers and Leaf Nodes

Initially, the Gnutella network (referred to as the *Gnet* from now on) was non-hierarchical. However, experience has shown that the abundance of signaling was a major threat to the scalability of the network [5]. Limewire (a company promoting an enhanced Gnutella servent[4]) suggested the introduction of a two-level hierarchy: *ultrapeers* (UPs) and *leaf nodes* (LNs). UPs are faster nodes in the sense that they are connected to high-capacity links and have a large amount of CPU power available. LNs maintain a single connection to their ultrapeer. A UP maintains 10-100 connections, one for each LN and 1-10 connections to other UPs [47]. The UPs do signaling on behalf of the LNs thus shielding them from large volumes of signaling traffic. A UP does not necessarily have leaf-nodes – it can work standalone.

Some servents may not be capable to become leaf nodes or ultrapeers for various reasons (e.g., they lack required functionality). In this case, they are labeled *legacy* nodes. In order to improve the overall scalability of the Gnet and to preserve bandwidth, UPs and LNs may refuse to connect to legacy nodes.

According to the Gnutella Development Forum (GDF) mailing list, the Gnutella community has recently adopted what is called support for high outdegree[5]. This implies that UPs maintain at least 32 connections to other UPs and 100–300 connections to different leaf nodes. LNs are recommended to maintain approximately connections to UPs. The numbers may differ slightly between different Gnutella vendors. The claim is that high-outdegree support allows a peer to

[3]This was discovered in the source code for gtk-gnutella-0.92. The software checks if the current date is later than July 1 2003. If true, it disables Gnutella v0.4 signaling.

[4]Servent denotes a software entity that acts both as a client or as a server. The name is a combination of the words **SERV**er and cli**ENT**.

[5]First mentioned in [48].

connect to the majority of Gnet peers in four hops or less.

3.2.2 Peer Signaling

Peer signaling can be divided into the following categories: peer discovery, resource query, ultrapeer routing and miscellaneous signaling. Peer discovery is done mainly through the use of Gnutella Web Cache (GWC) servers and PING and PONG messages. Query signaling consists of QUERY and QUERY_HIT messages. Ultrapeer routing can employ various schemes but the recommended one is the QRP. Ultrapeers signal among themselvs using PING and PONG messages. Finally, there are some miscellaneous messages flowing in the Gnet such as PUSH, Horizon Size Estimation Protocol (HSEP) or other messages based on proprietary Gnutella extensions.

3.2.3 Peer Discovery

A Gnutella node that wants to join the overlay must first have information about the listening socket[6] of at least another peer that is already member of the overlay. This is referred to as the *bootstrap problem*.

The old way to solve the bootstrap problem was to visit a web site that published up-to-date lists of known peers. The first step was to select one of the peers listed on the page, cut-and-paste its address (i.e., the listening socket) from the Web browser into the Gnutella servent and try to open a connection to it. This process would continue until at least one connection was successfully opened. At this point the PING–PONG traffic would, hopefully, reveal more peers to which the servent could connect. The addresses of newly found peers were cached in the local *hostcache* and reused when the servent application was restarted.

Since peers in general have a short life span [49] (i.e., they enter and leave the network very often) the hostcache kept by each node often got outdated. Gnutella Web Cache (GWC) servers[7] try to solve this problem. Each GWC server is essentially an HTTP server serving a list of active peers with associated listening sockets. The Web page is typically rendered by a Common Gateway Interface (CGI) script or Java servlet, which is also capable of updating the list contents. UPs update the list continuously, ensuring that new peers can always join the overlay.

A list of available GWC servers is maintained at the main GWebCache web site. This list contains only GWC servers that have elected to register themselves. Unofficial GWC servers exist as well.

New Gnutella peers implement the following bootstrap procedure: upon start they connect to the main GWebCache Web site, obtain the list of GWC systems, try to connect to a number of them, and finally end up building their own hostcache. Alternatively, the node can connect to an unofficial GWC system or connect directly to a node in the Gnet. The last option requires *a priori* knowledge about the listening socket of a Gnet node.

Recently, it was observed that GWC servers were becoming overloaded. There appeared to be two reasons behind the heavy load: an increase in the number of GWC-capable servents and the appearance of a large number of misbehaving servents. The UDP Host Cache (UHC) protocol was suggested as a way to alleviate the problem. The protocol works as a distributed bootstrap system, transforming UHC-enabled servents into GWC servers [50].

[6] By *socket*, we refer to the tuple <*host address, protocol, port*>.
[7] Also abbreviated as GWebCache servers.

3.2.4 Signaling Connection Establishment

Assuming a Gnutella servent has obtained the socket address (i.e., the IP address and port pair) of a peer, it will attempt to establish a full-duplex TCP connection. The explanation below will use typical TCP terminology calling the servent that has done the TCP active open *client* and its peer *server*. Once the TCP connection is in place, a handshaking procedure takes place between the client and the server:

1. The client sends the string `GNUTELLA CONNECT/0.6<CR><LF>` where `<CR>` is the ASCII code for carriage return and `<LF>` is the ASCII code for line feed.
2. The client sends all capability headers in a format similar to HTTP and ends with `<CR><LF>` on an empty line, e.g.,

```
User-Agent: BearShare/1.0<CR><LF>
X-Ultrapeer: True<CR><LF>
Pong-Caching: 0.1<CR><LF>
<CR><LF>
```

3. The server responds with the string `GNUTELLA/0.6 <status-code><status-string><CR><LF>`. The `<status-code>` follows the HTTP specification with code 200 meaning success. The `<status-string>` is a short human readable description of the status code (e.g., when the code is 200 the string will typically be set to OK).
4. The server sends all capability headers as described in step 2.
5. The client parses the server response to compute the smallest set of common capabilities available. If the client still wishes to connect, it will send `GNUTELLA/0.6 <status-code> <status-string><CR><LF>` to the server with the `<status-code>` set to 200. If the capabilities do not match, the client will set the `<status-code>` to an error code and close the TCP connection.

If the handshake is successful, the client and the server start exchanging binary Gnutella messages over the existing TCP connection. The existing TCP connection lasts until one of the peers decides to terminate the session. At that point the peer ending the connection has the opportunity to send an optional Gnutella BYE message. Then the peer closes the TCP connection.

Modern servents include a `X-Try` header in their response if they reject a connection. The header contains a list of socket addresses to recently active servents, to which the other peer can try to connect. The purpose of the `X-Try` header is to increase connectivity and reduce the need to contact a GWC server.

3.2.5 Compressed Message Streams

If the capability set used by the peers includes stream compression then all data on the TCP connection, with the exception of the initial handshake, is compressed [51]. The type of compression algorithm can be selected in the capability header, but the currently supported algorithm is *deflate*, which is implemented in zlib [52].

3.2.6 Gnutella Message Headers

Each Gnutella message starts with a generic header that contains the following:

- Message ID/GUID (Globally Unique ID) to uniquely identify messages on Gnet. Leaving out some details, the GUID is a mixture of the node's Ethernet MAC address and a timestamp [53].
- Payload type code that identifies the type of Gnutella message (e.g., PONG messages have payload type 0x01).
- Time-To-Live (TTL) to limit the signaling radius and its adverse impact on the network. Messages with TTL > 15 are dropped[8].
- Hop count to inform receiving peers how far the message has traveled (in hops).
- Payload length to describe the total length of the message following this header. The next generic Gnutella message header is located exactly this number of bytes from the end of this header.

The generic Gnutella header is followed by the actual message which may have its own headers. Also, the message may contain vendor extensions. Vendor extensions are used when a specific type of servent wants to implement experimental functionality not covered by the standard specifications. The vendor extensions should be implemented using Gnutella Generic Extension Protocol (GGEP) [54], since the protocol provides a transparent way for regular servents to interact with the vendor servents.

3.2.7 PING–ONG Messages

Each successfully connected pair of peers starts periodically sending PING messages to each other. The receiver of the PING message decrements the TTL in the Gnutella header. If the TTL is greater than zero the node increments the hop counter in the message header and then forwards the message to all its directly connected peers, with the exception of the one from where the message came. Note that PING messages do not carry any user data (not even the sender's listening socket). This means that the payload length field in the Gnutella header is set to zero.

PONG messages are sent only in response to PING messages. More than one PONG message can be sent in response to one PING. The PONG messages are returned on the *reverse path* used by the corresponding PING message. Each PONG message contains detailed information about *one* active Gnutella peer. It also contains the same GUID as the PING message that triggered it. The PONG receiver can, optionally, attempt to connect to the peer described in the message.

UPs use the same scheme, however they do *not* forward PINGs and PONGs to/from the LNs attached to them.

3.2.8 QUERY and QUERY_HIT Messages

A Gnutella peer wishing to locate some specific resource (e.g., file) must assemble a QUERY message. The message describes the desired resource using a text string. For a file resource this is the file name. In addition, the minimum speed (i.e., upload rate) of servents that should respond to this message is specified as well. There may be additional extensions attached to the message (e.g., proprietary extensions) but those are outside the scope of this document.

In Gnutella v0.4, the QUERY message is sent to all peers located one hop away, over the signaling connections established during the handshake. Peers receiving a QUERY message forward it to all directly connected peers unless the TTL field indicates otherwise.

[8]Nodes that support high outdegree will drop messages with TTL > 4.

The newer Gnutella v0.6 attempts to alleviate the problems of the previous version by introducing a form of selective forwarding called *dynamic query* [48]. A dynamic query first probes how popular the targeted content is. This is done by using a low TTL value in the QUERY message that is sent to a very limited number of directly connected peers. A large number of replies indicate popular content, whereas a low number of replies imply rare content. For rare content, the QUERY TTL value and the number of directly connected peers receiving the message are gradually increased. This procedure is repeated until enough results are received or until a theoretical limit of the number of QUERY message receivers is reached. This form of resource discovery requires all LNs to rely on UPs for their queries (i.e., LNs do not perform dynamic queries).

If a peer that has received the QUERY message is able to serve the resource, it should respond with a QUERY_HIT message. The GUID for the QUERY_HIT message must be the same as the one in the QUERY message that has triggered the response. The QUERY_HIT message lists each resource name that matches the resource description from the QUERY message[9] along with the resource size in bytes and other information. In addition, the QUERY_HIT messages contain the listening socket which should be used by the message receiver when it wants to download the resource. The Gnutella specification discourages the use of messages with sizes greater than 4 kB. Consequently, several QUERY_HIT messages may be issued by the same servent in response to a QUERY message.

The QUERY_HIT receiver must establish a direct HTTP connection to the listening socket described by the message (Section 3.2.13) in order to download the data. If the QUERY_HIT sender (i.e., the resource owner) is behind a firewall, incoming connections will typically not be accepted.

To work around this problem, when a firewall is detected, the downloader must send a PUSH message over the signaling connection. The message will be routed in reverse direction along the path taken by the received QUERY_HIT message. The resource owner can use the information in the PUSH message to establish a TCP connection to the downloader. The downloader can then use the HTTP GET method to retrieve the resource. For details, see Section 3.2.10.

Some servents use the metadata extension mechanism [55] to allow for richer queries. The idea is that metadata (e.g., author, genre, publisher) is associated with files shared by a servent. Other servents can query those files not only by file name, but also by the metadata fields.

3.2.9 Query Routing Protocol

The mission of ultrapeers is to reduce the burden put on the network by peer signaling. They achieve this goal by eliminating PING messages among leaf nodes and by employing query routing. There are various schemes for ultrapeer query routing but the recommended scheme the QRP [56]. Ultrapeers signal among themselves by using PING and PONG messages.

QRP [56] was introduced in order to mitigate the adverse effects of flooding used by the Gnutella file queries and is based on a modified version of *Bloom filters* [57]. The idea is to break a query into individual keywords and have a hash function applied to each keyword. Given a keyword, the hash function returns an index to an element in a finite discrete vector. Each entry in the vector is the minimum distance expressed in hops to a peer holding a resource that matches the keyword in the query. Queries are forwared *only* to leaf nodes that have resources that match *all* the keywords. This substantially limits the bandwith used by queries. Peers run the hash algorithm over the resources they share and exchange the routing tables (i.e., hop vectors) at regular intervals.

[9] For example the string `linux` could identify a resource called `linux_redhat_7.0.iso` as well as a resource called `linux_installation_guide.txt.gz`. Thus, this query would yield two potential results. Both results will be returned to the QUERY sender.

Individual peers (legacy or ultrapeer nodes) may run QRP and exchange routing tables among themselves [58]. However, the typical scenario is that legacy nodes do not use QRP, leaf nodes send route table updates only to ultrapeers, and ultrapeers propagate these tables only to directly connected ultrapeers.

3.2.10 PUSH Messages

PUSH messages are used by peers that want to download resources from peers located behind firewalls that prevent incoming TCP connections. The downloader sends a PUSH message over the existing TCP connection, which was setup during the handshake phase. The PUSH message contains the listening socket of the sender. The host behind the firewall can then attempt to establish a TCP connection to the listening socket described in the message. If the TCP connection is established successfully, the host behind the firewall sends the following string over the signaling connection:

```
GIV <File Index>:<Servent Identifier>/<File Name><LF><LF>
```

The `<File Index>` and `<Servent Identifier>` are the values found in the corresponding PUSH message and `<File Name>` is the name of the resource requested. Upon the receipt of the message the receiver issues an HTTP GET request on the newly established TCP connection.

```
GET /get/<File Index>/<File Name> \
HTTP/1.1<CR><LF>
User-Agent: Gnutella<CR><LF>
Connection: Keep-Alive
Range: bytes=0-<CR><LF>
<CR><LF>
```

3.2.11 BYE Messages

The BYE message is an *optional* message used when a peer wants to inform its neighbours that it will close the signaling connection. The message contains an error code along with an error string. The message is sent only to hosts that have indicated during handshake that they support BYE messages.

3.2.12 Horizon Size Estimation Protocol Messages

The Horizon Size Estimation Protocol (HSEP) is used to obtain estimates on the number of reachable resources (i.e., nodes, shared files and shared kilobytes of data) [59]. Hosts that support HSEP announce this as part of the capabiliy set exchange during the Gnutella handshake. If the hosts on each side of a connection support HSEP, they start exchanging HSEP message approximately every 30 seconds. The HSEP message consists of `n_max` triples. Each triple describes the number of nodes, files and kilobytes of data estimated at the corresponding number of hops from the node sending the message. The `n_max` values is the maximum number of hops supported by the protocol with 10 hops being the recommended value [59].

The horizon size estimation can be used to quantify the quality of a connection: the higher the number of reachable resources, the higher the quality of the connection.

3.2.13 Data Exchange (File Transfer)

Data exchange takes place over a direct HTTP connection between a pair of peers. Both HTTP 1.0 and HTTP 1.1 are supported but use of HTTP 1.1 is strongly recommended. Most notably, the use of features such as persist connection and range request is encouraged.

The range request allows a peer to continue an unfinished transfer from where it left off. Furthermore, it allows servents to utilize *swarming*, which is the technique to retrieve different parts of the file from different peers. Swarming is not part of the Gnutella protocol and regular Gnutella servents (i.e., servents that do not explicitly support swarming) can be engaged in swarming without being aware of it. From their point of view, a peer is requesting a range of bytes for a particular resource. The intelligence is located at the peer downloading data.

The persist connection feature is useful for swarming. It allows a peer to make several requests for different byte ranges in a file, over the *same* HTTP connection.

Fig. 3.5 shows a simple Gnet scenario, involving three legacy peers. It is assumed that Peer A has obtained the listening socket of Peer B from a GWC server. Using the socket descriptor, Peer A attempts to connect to Peer B. In this example, Peer B already has a signaling connection to Peer C.

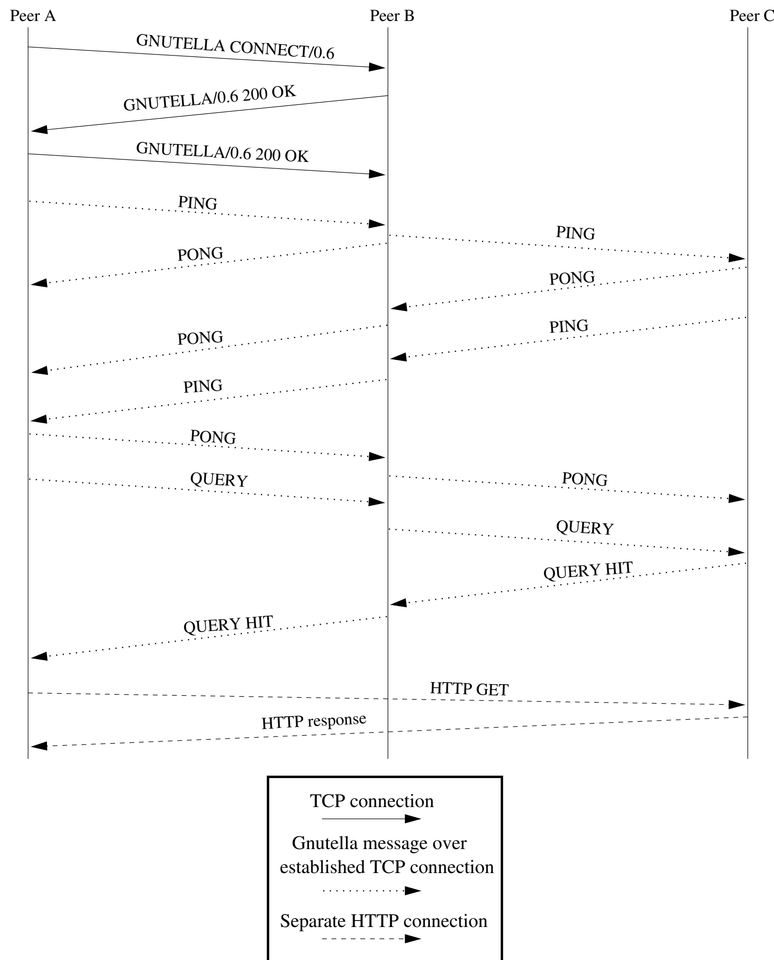


Figure 3.5: *Example of a Gnutella session.*

The first three messages between Peer A and Peer B illustrate the establishment of the signaling connection between the two peers. The two peers may exchange capabilities during this phase as well.

The next phase encompasses the exchange of network topology information with the help of PING and PONG messages. The messages are sent over the TCP connection established previously (i.e., during the peer handshake). It is observed that PING messages are forwarded by Peer B from Peer A to Peer C in both directions as well as that PONG messages follow the reverse path taken by the corresponding PING message.

At a later time the Peer A sends a QUERY message, which is forwarded by Peer B to Peer C. In this example, only Peer C is able to serve the resource, which is illustrated by the QUERY_HIT message. The QUERY and QUERY_HIT messages use the existing TCP connection, just like the PING and PONG messages. Again, it is observed that the QUERY_HIT message follows the reverse path taken by the corresponding QUERY message.

Finally, Peer A opens a direct HTTP connection to Peer C and downloads the resource by using the HTTP GET method. The resource contents are returned in the HTTP response message.

The exchange of PING–PONG and QUERY–QUERY_HIT messages continues until one of the peers tears down the TCP connection. A Gnutella BYE message may be sent as notification that the signaling connection will be closed.

3.2.14 Other Features

Gnutella has support for many other featues which are important, but not whithin the scope of this document. The remainder of this section will present some of these features briefly.

Hash/URN Gnutella Extensions

The Hash/URN Gnutella Extensions (HUGE) specification [60] provides a way to identify files by Uniform Resource Names (URN) and in particular by SHA-1 hash values. The advantages of using HUGE and SHA-1 is that files with same content but different names can be discovered through the QUERY–QUERY_HIT mechanism and that the file integrity can be checked upon download by recomputing the SHA-1 hash value.

File Magnets and Magma Lists

Building on HUGE, file magnets represent the bridge between the Web and P2P networks. Web pages can include special URL links (file magnets), which encode URNs to resources available on the P2P network. When a user clicks on such a link, the web browser will transfer the URN to the local Gnutella servent, which will perform a query on Gnet.

A *Magma list* is a list of file magnets, e.g., the favourite documents, music or pictures shared by a Gnutella user.

Download Mesh

In order to speed up file downloads and to distribute the load among servents, when a peer sends a QUERY_HIT message it includes a list of peers that are known to have the same file (i.e.,

the download mesh). The simplest way a servent can build such a list is to remember the list it obtained itself when it downloaded the file. Download meshes require support for the HUGE extension. The main benefit of using a download mesh is the ability to perform efficient swarming, i.e., to be able to quickly download simultaneously from several locations.

Partial File Sharing

Partial File Sharing (PFS) is an optimization of swarming and download meshes. Servents that support PFS do not wait to download the whole file before replying to matching QUERY messages. If a servent requests the file before its download has completed, the servent that has the partial file will set the `Content-Range` header in the HTTP reply informing the other peer about the amount of available data.

Passive/Active Remote Queueing

Most servents limit the number of uploads that can occur simultaneously, in order to preserve bandwidth. When Passive/Active Remote Queueing (PARQ) is used, download requests are queued at the servent hosting the file [61]. The specification allows servents to check their place in the download queue. It allows hosts to temporarily (maximum 5 minutes) become unavailable. This feature can come in handy if the servent crashes or temporarily looses its Internet connection.

Firewall-to-Firewall aka. Reliable UDP File Transfer

Firewall-to-Firewall (F2F) allows two firewalled hosts, both of them connected to Internet through NAT servers, to transfer files among themselves over UDP [62]. The technique to open the UDP ports in the firewall is known as “UDP hole punching” and is documented fairly well in [63].

LAN Multicast

Recently, a specification was made available which allows servents located on the same LAN to take advantage of IP multicast to transfer files [64]. The advantages of LAN multicast is that file transfers are more efficient due to IP multicast and generally much faster due to higher bandwitdh, lower latency and lower number of hops.

Chapter 4

Measurement Software Description

Traffic measurements have been used within the networking research community for a very long time, going back to the early teletraffic researchers, such as Conny Palm and Engset [65].

Williamson [66] identifies four main reasons for the usefulness of network traffic measurements: network troubleshooting, protocol debugging, workload characterization and performance evaluation. For the present work, only the latter two are considered, with an emphasis on workload characterization.

When deciding on making measurements, there are two avenues from which to choose: active or passive measurements. Active measurements entail actively probing a network with either artificially generated traffic or having a node join in the network as an active participant. Probing with artificial traffic is analogous to system identification using impulses in, e.g., vibration experiments or acoustical environments. A passive measurement is one where the network is silently monitored without any intrusion.

4.1 Passive measurements

Passive measurements are commonly used when data on "real" networks are desired, for instance to use in trace-driven simulations, model validation or bottleneck identification. Essentially, this technique observes a live network without interacting with it.

Depending on the level of accuracy desired, different measurement options are available. For coarse-grained measurements, on a timescale the order of seconds, there is the possibility of using SNMP and RMON to gather information from networking hardware. This is usually used as part of normal network operations, and not very useful for protocol evaluations and per-flow performance evaluation. Per-flow information is available in, e.g., Cisco's NetFlow [67] , but again, no packet inspection is available.

Finer-grained measurements with full packet inspection capabilities are available in both hardware and software configurations. Software configurations provide measurement accuracies in tens of microseconds, while dedicated measurement hardware gives nanosecond accuracy.

There are two main approaches to perform passive application layer measurements for network traffic. In the first approach, called *application logging*, the traffic is measured by the application itself. The other approach is to obtain measurements indirectly, by monitoring traffic at the link layer and performing application flow reassembly using a specially designed application. This

approach is referred to as *flow reassembly*. A mixture of these two approaches is possible as well.

4.1.1 Application Logging

Unless already supported, application logging requires changes in the application software to record incoming and outgoing network data and other events of interest, e.g., transitions between application states, failures, CPU and memory usage etc. This implies modifying the application source code. In the case of open source software these changes can be performed rather straightforwardly. However, for closed source software one would need to negotiate an agreement with the vendor to obtain and modify the source code.

The advantage of application logging is that measurement data is readily available from within the application. Measurement code embedded at relevant places in the application can continuously monitor all variables of interest. On the other hand, the main disadvantage associated with this method (apart from the licensing issue discussed above) is related to timestamp accuracy. The accuracy of the timestamp is affected by three main factors: drift in the frequency of the crystal controlling the system clock, latency in the Operating System (OS) and latency in the network stack.

The frequency drift of the crystal is due to temperature changes and age. Its influence on the timestamp is rather small when compared to the other two factors.

Latency in the OS refers to the delay between the time when a user-space process requests a timestamp from the operating system and the time when the timestamp is available to the process. The delay is largely accounted for by scheduling in the kernel when the calling process is temporarily preempted by other processes. The problem becomes increasingly worse for interpreted programs, e.g., the reference BitTorrent client, which is a python script. In this case the timestamps are subject to additional scheduling imposed by the interpreter.

A significant amount of queueing and scheduling occurs in the TCP/IP stack as well, especially in the routines for IP and TCP reassembly. The effect is that timestamps at the application layer are only indicative for the actual time when packets enter or leave the link-layer at the node in question.

4.1.2 Flow Reassembly

The flow reassembly method attempts to address some of these problems by moving the measurements closer to the network. Link-layer measurements have enjoyed a long tradition in the network community. However, since the interest is now moving towards what happens at the application layer, one needs to develop dedicated software able to decode application layer messages from the observed link-layer traffic, essentially replicating parts of the application of interest.

Flow reassembly involves three stages: link-layer capture, transport stream reassembly, e.g., TCP reassembly, and application message decoding.

A plethora of link-layer capture software is available under very liberal licenses on the Internet, e.g., tcpdump and ethereal [68, 69]. The common denominator for this software is that in a shared-medium LAN such as Ethernet, the capturing software forces the network interface to work in promiscuous mode, thus enabling it to monitor all traffic in the LAN. An issue to consider carefully when selecting capture software, is the timestamping operation. The operation should be performed as close as possible to the place where the frame is read from the network card (if possible in the network driver or on the card itself). Failure to do so may lead to similar inaccuracies as in the case of application logging.

Transport stream reassembly deals with missing or duplicate packets and with packets arriving in the wrong order. At the IP layer this involves reassembly of IP fragments. At the TCP layer, the transport stream reassembly replicates the TCP reassembly functionality from the network stack.

The main problem with regards to TCP reassembly is to obtain the same TCP state transitions that occurred at the time when the traffic was recorded. This is particularly hard to do in a heterogeneous network environment, since different OSs handle special TCP conditions in different ways. For example, retransmitted segments may overlap the data received previously and one must decide whether to keep the old or the new data. Windows and UNIX take opposing views of this scenario [70]. A solution to minimize the inconsistencies in protocol implementations is to use a traffic normalizer [71]. Similar problems apply to reassembly of IP fragments.

Application message decoding uses reassembled transport layer flows to obtain application messages exchanged by end-points. The main advantage of flow reassembly is that it provides a more accurate view of how the application affects the network. Furthermore, flow reassembly can be run on a dedicated host different from the hosts participating in the application session. Such a dedicated host has also the possibility to analyze all traffic passing by the recording interface. In contrast, application logging can only provide information about the flows in which the measuring host is an active participator. Furthermore, the flow reassembly method can save link-layer traffic to disk for off-line analysis.

A major disadvantage associated with flow reassembly is that all application states must be inferred from the recorded network traffic. This is not always possible, since certain application state transitions may be independent of network events. Another disadvantage is that a lot of existing functionality (e.g., IP and TCP reassembly) is duplicated. A well-known programming mantra states that the probability to encounter bugs increases proportionally to the volume of new code. An even more serious problem is related to the link-layer capture. On heavily loaded links, the hardware may not be able to record all data and will start dropping frames. This has an impact on the host performing the measurements but not necessarily on the host participating in the application layer session.

Off-line traffic analysis features, similar to those found in flow reassembly, can be implemented using application logging by adding suitable message recording points in the software application. This means, in fact, a measurement method that is a mixture between application logging and flow reassembly. Such a mixed methodology has the advantages of both methods, e.g., no need to infer application state from link-layer traces, and no need to decide beforehand what statistics to collect.

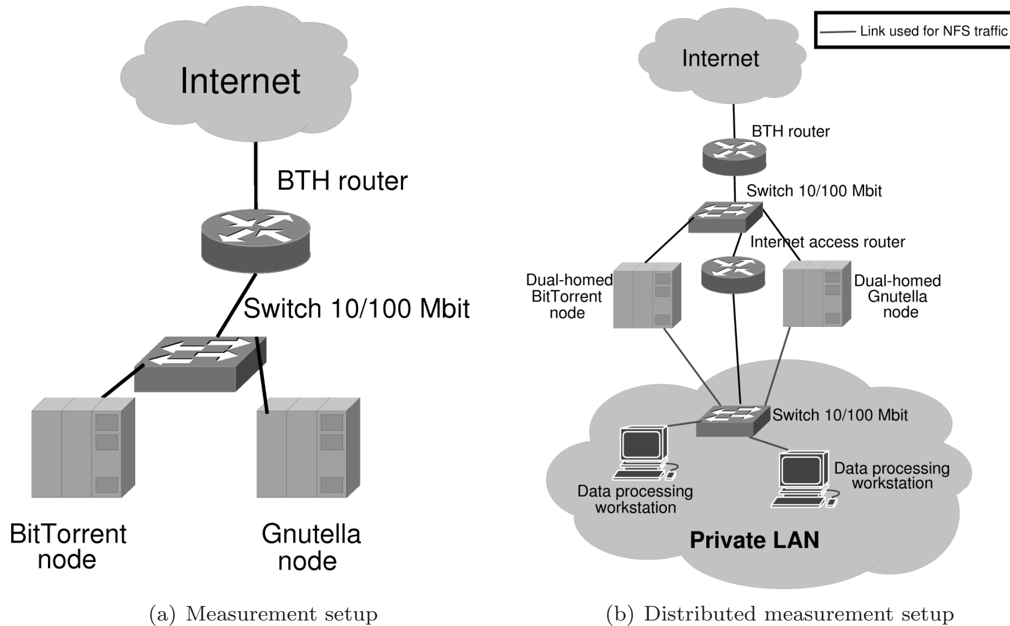
4.2 Network infrastructure

The P2P measurement infrastructure developed at BTH consists of peer nodes and protocol decoding software. Tcpdump [68] and tcptrace [72] are used for traffic recording and protocol decoding. Although the infrastructure is currently geared towards P2P protocols, it can be easily extended to measure other protocols running over TCP as well. Furthermore, we plan to develop similar modules to measure UDP-based applications as well.

The BTH measurement nodes run the Gentoo Linux 1.4 operating system, with kernel version 2.6.5. Each node is equipped with an Intel Celeron 2.4 GHz processor, 1 GB RAM, 120 GB hard drive, and 10/100 FastEthernet network interface. The network interface is connected to a 100 Mbit switch in the lab at the Telecommunication Systems department, which is further connected through a router to the GigaSUNET backbone (Fig. 4.1(a)).

Our experience with the current setup has been that the traffic recording step alone accounts for

about 70 % of the total time taken by measurements. Protocol decoding is not possible when the hosts are recording traffic. The main reason is that the protocol decoding phase, which is I/O intensive, requires large amounts of CPU power and RAM. To overcome this problem, we are proposing a distributed measurement infrastructure similar to the one shown in Fig. 4.1(b).



(a) Measurement setup

(b) Distributed measurement setup

Figure 4.1: *Measurement network infrastructures.*

When used in the distributed infrastructure, the P2P nodes are equipped with an additional network interface, which we refer to as the *management* interface. P2P traffic is recorded from the primary interface and stored in a directory on the disk. The directory is exported using the Network File System (NFS) over the management interface. Data processing workstations can read recorded data over NFS as soon as it is available. Optionally, the data processing workstations can be located in private LAN or VPN in order to increase security, save IP address space and decrease the number of collisions on the Ethernet segment. In this case, the Internet access router provides Internet access to the workstations, if needed.

4.3 TCP Reassembly Framework

Each measurement node has tcpdump version 3.8.3 installed on it. When the node is running measurements, tcpdump is started before the Gnutella servent in order to avoid missing any connections. Tcpdump can also be run on a different node in the network, provided that the ultrapeer switch port is mirrored to the port where the tcpdump host is recording or if the switch is replaced with a hub and both the tcpdump host and the ultrapeer are connected to it.

During the data collection stage, tcpdump collects Ethernet frames from the switch port where the ultrapeer node is connected. Since most P2P applications can use dynamic ports, all traffic reaching the switch port must be collected. However, to increase the performance during data collection and data processing, one can turn off most or all server software on the ultrapeer node. It is possible, in addition, to apply a filter to tcpdump that drops packets used by traditional services, which are running on well-known ports (e.g., HTTP, FTP, SSH).

The volume of collected data can be quite large, e.g., the resulting trace file could grow well beyond 2 GB in less than one day, which is larger than most standard filesystems can handle without modification. This is directly related to the number of peers the servent is allowed to connect to. In the case of Gnutella we observed on average 130 peers (100 leaf nodes and 30 ultrapeers) and collected approximately 33 GB captured trace data in eleven days. The solution was to have tcpdump spread the recorded data across several files, each 600 MB large. This file size was chosen such that each data file is small enough to fit on a recordable CD.

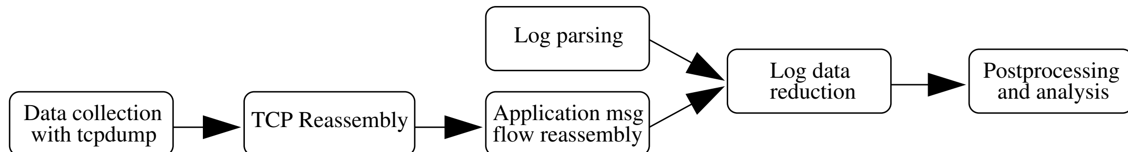


Figure 4.2: *Measurement procedures.*

4.3.1 TCP Reassembly

Assuming that the measured P2P application runs over TCP, the next step is to reassemble the TCP frames to a flow of ordered bytes. The TCP reassembly module builds on the TCP engine available in tcptrace.

The module reads the tcpdump traces in the order they were created. Each trace is scanned for TCP connections. When found, they are stored in a list with connection records. Further, when a new TCP segment is found in the trace file, the module scans the connection list comparing the socket pair of the segment with each entry in the list. If no entry matches the socket pair of the new segment, then a new connection is considered to be found and a record is created for it, which finally is added to the connection list. Otherwise, the connection record matching the socket pair is retrieved and sent together with the new segment to the TCP reassembly engine.

The TCP reassembly engine is similar to the one used by the FreeBSD TCP/IP stack as described in [73]. For each active connection, the reassembly engine keeps a doubly linked list, which is referred to as the reassembly list. When given a connection record and a new segment, it retrieves the correct reassembly list and then it inserts the new segment in the correct place in the list. The reassembly engine is capable of handling out-of-order segments as well as forward and backward overlapping between segments.

4.3.2 Application Data Flow Reassembly

Whenever new data is available, the application data reassembly module is notified. Upon notification, it will ask the TCP reassembly module for a new segment from the reassembly list corresponding to the socket pair received with the notification. When it receives the new segment, it interprets the contents according to the specification for the protocol it decodes. Since application messages may span several segments and since a segment may contain data from two consecutive messages, each segment is appended to the end of a data buffer before further processing, thus creating a contiguous data flow containing at least one application message.

Gnutella Reassembly

In the case of a new Gnutella connection, the application reassembly module first waits for the handshake phase to begin. If the handshake fails the connection is marked invalid and it is eventually discarded by the memory manager.

If the handshake is successful, the application reassembly module scans the capability lists sent by the nodes involved in the TCP connection. If the nodes have agreed to compress the data, the connection is marked as compressed. Further segments received from the TCP reassembly module for this connection are first sent to the decompressor, before being appended to the data buffer.

The decompressor uses the *inflate()* function of zlib [52] to decompress the data available in the new segment. Upon successful decompression the decompressed data is appended to the data buffer.

Immediately after the handshake phase, the application reassembly module attempts to find the Gnutella message header of the first message. Using the payload length field, it is able to discover the beginning of the second message. This is the only way in the Gnutella protocol to discover message boundaries and thus track application state changes. Based on the message type field in the message header, the corresponding decoding function is called, which outputs a message record to the log file. The message records follow a specific format required by the postprocessing stage.

BitTorrent Reassembly

The BitTorrent reassembler works in a fashion much similar to the Gnutella reassembler. It is however less complex, since the BitTorrent protocol is substantially less complex than the Gnutella protocol. Each BitTorrent message is fixed-length and is prepended by the message type in a 32-bit little-endian word, making the decoding straightforward.

The timestamp of the first TCP segment of each message is recorded, along with the timestamp of the last segment. In the case of single-segment messages (all messages except the *piece*-messages), the first and last segments are the same.

A rudimentary HTTP parser is also available, which is used to parse tracker responses.

4.3.3 Data Compression and Postprocessing

Since the logs can grow quite large, they can be processed through an optional stage of data compression. The compression is achieved by using the on-the-fly *deflate* compression offered by zlib. Additional data reduction can be achieved if the user is willing to sacrifice some detail by aggregating data over time.

The postprocessing module interprets the (optionally compressed) log data and it is able to demultiplex it based on different types of constraints: message type, IP address, port number, etc. The data output format of this stage is suitable for input to numerical computation software such as MATLAB and standard UNIX text processing software such as sed, awk and perl.

4.4 Application Logging

Application logging is commonly used in server software to enable traceability of errors and client requests. In certain server applications, such as critical business systems and other high-security systems, server logs are very important for detecting intrusion attempts and for estimating severity of security breaches. In other applications, logs are a useful tool for performance analysis.

However, client applications do not usually provide for much in terms of logging. If logging is made available, it usually provides rather coarsely grained information, such as application start and other very high-level application events. It is unusual that an application provides the amount of log detail needed to analyze the network performance of the application.

To provide adequate detail in application logs, it is necessary to modify the application in such a way that the application both provides the detailed event information needed and a way to store this information in a log file or database.

In applications that are based on an event-loop with a central managing component, obtaining the relevant information is a fairly easy task, as the events being handled contain all information relevant to the specific event. By adding a timestamp, these may then be ejected to a log file or database. On the other hand, in a threaded and less centralised application, this becomes a more difficult task, as events may not be handled through a single component.

An additional issue with client-side logging is deployment of the modified clients. It is important to have a large enough number of users to provide representative data. Also, not all users may agree to running a modified client.

One of the most difficult problems relates to the non-availability of client-source code. For example, most proprietary software does not provide the source code for the application, making modification impossible without substantial reverse engineering.

Log storage may become an issue if, for instance, the application is running on an embedded system where there is no storage available except for internal memory. Also, if measurements are performed over a long period of time and/or there is a large number of events, the application logs may grow prohibitively large.

4.4.1 BitTorrent Software Modifications

The reference BitTorrent client version[1] is written in the python programming language [74]. Python is an interpreted and interactive language with object oriented features that combines syntactical clarity with powerful components and system-level functionality. This makes the process of extending software written in the language less complicated than in a compiled and syntactically more demanding language such as C or Java.

The client is written as an event-based program, reacting on incoming protocol messages and internal timers. The internal timers activate the sending of messages such as tracker requests, unchoking peers and network timeouts. For the purpose of the present work, the incoming network message handling routines are the important part. These are mainly located in a single software component, which handles all incoming events. This component consists of a function containing the main loop (that receives the network messages), and several message specific functions to handle the incoming messages that are invoked from the main loop. While it is possible to intercept the messages in the main loop, it is much easier to do so in the specific message handling routines. There are two major reasons for this:

[1] Version 3.4.1, released on March 11, 2004.

- The message type is already implicitly given by the call of the function.
- Message-specific information is provided automatically, without the need to write extra parsing code. For instance, in the case of a piece request message being intercepted in the main loop, it would have been necessary to parse the incoming message to find information such as piece number and subpiece index.

Before saving the ejected log messages to disk, they are compressed by the zlib library [52]. This is beneficial both with regards to disk storage and with regards to the amount of disk I/O performed. The degradation in CPU performance of the compression is practically negligible on the measurement computers.

Finally, extra parameters have been added to the application to allow changing the filename of the log-file, and code to automatically generate a date and timestamped filename if none was given.

4.5 Log Formats

Selection of a log format that provides a suitable amount of information is an important issue. It is important to capture enough information to make relevant statistical analysis possible, while at the same time keep the sizes of the log files to a manageable level.

This problem is most noticeable when designing a log format for application logs, as it is not possible to re-run a specific measurement a second time if one has chosen too small a subset of metrics to log. Packet captures are less affected by this, but are not impervious to similar effects in the case of, for instance, too small capture size for the recorded packets, thus losing parts of the payload data. In both cases, information is irretrievably lost.

Complete packet captures that contain all data transmitted on a link may be used to re-generate log files as needed. This is however often a very time-consuming process, and it is preferable to avoid it whenever possible.

4.5.1 BitTorrent XML Log Format

The eXtensible Markup Language (XML) [75] has a number of attractive features that makes it a good choice as a log format. XML is by concept and design made to be easily parsable by a computer, while at the same time be at least semi-readable by humans.

Some of the salient advantages of using XML as a log format are :

Parsability There are several XML parsing libraries available for a plethora of languages, including, but not limited to, perl, C, C++, python and MATLAB. This makes the writing of log parsers much easier, since it is not necessary to write an application specific parser for the log format.

Extensibility It is easy to add new log fields, and new log fields do not necessitate changing the parser. This is very useful when deciding what information goes into the log, as fields may be added and removed easily.

Validation The number and types of fields are easily verifiable, and is usually performed as part of the XML validation process provided by the parsing library.

Two drawbacks with using XML as a log format are that the parsing is slightly slower than an application specific parser, and that memory requirements are substantially higher when using specific parsers. In particular, it is rarely possible to use the Document Object Model (DOM) parsers to parse the log files. These parsers maintain a representation of the entire XML file in memory and, with log files in the gigabyte range, the amount of memory required is substantial. Simpler parsers, such as Simple API for XML (SAX) parsers, are therefore used. These parse the document on an element by element basis, removing the need for keeping the entire document in memory. This solution unfortunately also means that the transformation capabilities provided by eXtensible Stylesheet Language Transformations (XSLT) cannot be used, and specific software making use of provided SAX parsers must be created. A third drawback is that using XML adds metadata, which in turn makes the storage requirements for the logs higher.

XML documents are text documents comprised of *elements* and *attributes*. Attributes are contained within the elements, and usually carry element-specific information and modifiers. The XML document type used for the BitTorrent log files is comprised of only two elements: `EVENTLIST` and `EVENT`. The `EVENTLIST` element carries information regarding the torrent-file used for the measurement and the settings that were used for the BitTorrent client during the measurement session. Figure 4.3 shows two excerpts from such an XML document.

Every `EVENT` element contains the attributes `type` and `timestamp`. The `timestamp` attribute signifies the time at which this event was ejected to the log file, expressed as a UNIX timestamp, i.e., the number of seconds elapsed since 00:00:00 UTC, January 1, 1970. The `type` field denotes the event type. The various values for the `type`-attribute are:

announce The only tracker-related event type available. It is ejected into the log file when the peer communicates with the tracker to request more peers. This element carries the following attributes:

- **uploaded** Denotes the number of subpiece bytes this peer has sent to other peers since it was launched.
- **downloaded** Denotes the number of subpiece bytes this peer has received from other peers since the client was launched.
- **left** Denotes the number of bytes of the resource that remain to download.
- **last** This parameter is undocumented in both the official protocol specification and Wiki.
- **trackerid** Used by the tracker for maintaining state.
- **event** Is one of `started`, `none` or `completed`. The value `started` should be used when sending the initial tracker `announce` message, and only then. The `None` value is used when transmitting the periodic updates to the tracker, while the value `completed` is sent exactly once to the tracker when the download is complete.
- **numwant** Denotes the number of new peer addresses the peer is requesting from the tracker.

start_dl This element is ejected for every newly initiated TCP connection to a peer. Note that it does not necessarily imply that the BitTorrent handshake will be completed.

connect This element is ejected after every completed BitTorrent handshake.

unchoke, choke, interested, not interested, request, piece, have, cancel

```
<EVENT type="announce" timestamp="1084428217.041880" uploaded="0" downloaded="0" left="661127168" last="None" trackerid="None" event="started" numwant="50"/>
<EVENT type="start_dl" timestamp="1084428218.717552" src="220.233.6.19" port="6881" nconns="1"/>
<EVENT type="unchoke" timestamp="1084428218.717747" dst="220.233.6.19" dstid="-AZ2084-v8j2jYQi0GOq" nconns="1"/>
<EVENT type="connect" timestamp="1084428218.717867" src="220.233.6.19" srcid="-AZ2084-v8j2jYQi0GOq" nconns="1"/>
<EVENT type="bitfield" timestamp="1084428219.388488" src="220.233.6.19" srcid="-AZ2084-v8j2jYQi0GOq" nconns="1"/>
<EVENT type="interested" timestamp="1084428219.429775" dst="220.233.6.19" dstid="-AZ2084-v8j2jYQi0GOq" nconns="1"/>
<EVENT type="start_dl" timestamp="1084428225.820689" src="217.226.127.188" port="6881" nconns="2"/>
<EVENT type="unchoke" timestamp="1084428225.820863" dst="217.226.127.188" dstid="-AZ2084-8wtClbR51oMR" nconns="2"/>
<EVENT type="connect" timestamp="1084428225.820989" src="217.226.127.188" srcid="-AZ2084-8wtClbR51oMR" nconns="2"/>
<EVENT type="unchoke" timestamp="1084428229.588582" src="220.233.6.19" srcid="-AZ2084-v8j2jYQi0GOq" nconns="2"/>
<EVENT type="request" timestamp="1084428229.588833" dst="220.233.6.19" dstid="-AZ2084-v8j2jYQi0GOq" piece="688" begin="0" nconns="2"/>
<EVENT type="request" timestamp="1084428229.589048" dst="220.233.6.19" dstid="-AZ2084-v8j2jYQi0GOq" piece="688" begin="16384" nconns="2"/>
<EVENT type="request" timestamp="1084428229.589190" dst="220.233.6.19" dstid="-AZ2084-v8j2jYQi0GOq" piece="688" begin="32768" nconns="2"/>
<EVENT type="request" timestamp="1084428229.589342" dst="220.233.6.19" dstid="-AZ2084-v8j2jYQi0GOq" piece="688" begin="49152" nconns="2"/>
<EVENT type="request" timestamp="1084428229.589477" dst="220.233.6.19" dstid="-AZ2084-v8j2jYQi0GOq" piece="688" begin="65536" nconns="2"/>
<EVENT type="bitfield" timestamp="1084428230.003638" src="217.226.127.188" srcid="-AZ2084-8wtClbR51oMR" nconns="2"/>
<EVENT type="interested" timestamp="1084428230.052099" dst="217.226.127.188" dstid="-AZ2084-8wtClbR51oMR" nconns="2"/>
<EVENT type="start_dl" timestamp="1084428230.842135" src="67.70.42.140" port="6881" nconns="3"/>
<EVENT type="unchoke" timestamp="1084428230.842327" dst="67.70.42.140" dstid="-AZ2084-11G41aBmxdcr" nconns="3"/>
<EVENT type="connect" timestamp="1084428230.842438" src="67.70.42.140" srcid="-AZ2084-11G41aBmxdcr" nconns="3"/>
<EVENT type="piece" timestamp="1084428235.373197" src="220.233.6.19" srcid="-AZ2084-v8j2jYQi0GOq" piece="688" begin="0" length="16384"/>
<EVENT type="request" timestamp="1084428235.375262" dst="220.233.6.19" dstid="-AZ2084-v8j2jYQi0GOq" piece="688" begin="81920" nconns="3"/>
<EVENT type="bitfield" timestamp="1084428238.242941" src="67.70.42.140" srcid="-AZ2084-11G41aBmxdcr" nconns="3"/>
<EVENT type="interested" timestamp="1084428238.279325" dst="67.70.42.140" dstid="-AZ2084-11G41aBmxdcr" nconns="3"/>
<EVENT type="piece" timestamp="1084428238.983001" src="220.233.6.19" srcid="-AZ2084-v8j2jYQi0GOq" piece="688" begin="16384" length="16384"/>
<EVENT type="request" timestamp="1084428238.983420" dst="220.233.6.19" dstid="-AZ2084-v8j2jYQi0GOq" piece="688" begin="98304" nconns="3"/>
<EVENT type="piece" timestamp="1084428243.061652" src="220.233.6.19" srcid="-AZ2084-v8j2jYQi0GOq" piece="688" begin="32768" length="16384"/>
<EVENT type="request" timestamp="1084428243.062012" dst="220.233.6.19" dstid="-AZ2084-v8j2jYQi0GOq" piece="688" begin="114688" nconns="3"/>
<EVENT type="have" timestamp="1084428243.807769" src="67.70.42.140" srcid="-AZ2084-11G41aBmxdcr" piece="2325" nconns="3"/>
<EVENT type="piece" timestamp="1084428246.762416" src="220.233.6.19" srcid="-AZ2084-v8j2jYQi0GOq" piece="688" begin="49152" length="16384"/>
<EVENT type="request" timestamp="1084428246.762847" dst="220.233.6.19" dstid="-AZ2084-v8j2jYQi0GOq" piece="688" begin="131072" nconns="3"/>
<EVENT type="have" timestamp="1084428250.036548" src="67.70.42.140" srcid="-AZ2084-11G41aBmxdcr" piece="2376" nconns="3"/>
<EVENT type="have" timestamp="1084428250.036794" src="67.70.42.140" srcid="-AZ2084-11G41aBmxdcr" piece="2181" nconns="3"/>
.
.
.
<EVENT type="cancel" timestamp="1084430940.120284" dst="200.185.78.6" dstid="-AZ2084-a3QFRyfkLC8Q"  piece="1762" nconns="15"/>
<EVENT type="have" direction="out" timestamp="1084430940.120433" piece="1762" nconns="15" down="661110784"/>
<EVENT type="piece" timestamp="1084430940.129599" src="212.100.224.105" srcid="\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\xda\xe3.\xdb\x8b\xcfb\xa8"
       piece="1176" begin="49152" length="16384"/>
<EVENT type="announce" timestamp="1084430940.133816" uploaded="22118400" downloaded="661127168" left="0" last="None" trackerid="None" event="completed" numwant="50"/>
<EVENT type="done" timestamp="1084430940.134094" src="212.100.224.105" srcid="\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\xda\xe3.\xdb\x8b\xcfb\xa8"
       piece="1176" rxtime="0.645374" rxstart="1084430939.488720" nconns="15"/>
<EVENT type="cancel" timestamp="1084430940.134308" dst="220.233.6.19" dstid="-AZ2084-v8j2jYQi0GOq"  piece="1176" nconns="15"/>
<EVENT type="not interested" timestamp="1084430940.134540" dst="220.233.6.19" dstid="-AZ2084-v8j2jYQi0GOq" nconns="15"/>
<EVENT type="not interested" timestamp="1084430940.134640" dst="67.70.42.140" dstid="-AZ2084-11G41aBmxdcr" nconns="15"/>
<
```


Figure 4.3: *Extract from BitTorrent XML log file.*

These element types are ejected for each sent or received corresponding BitTorrent protocol message.

send The send-element is the equivalent of the piece-message, but for the subpieces the local peer transmits.

done This element is ejected once a download completes fully, and should only appear once per log file.

The various peer-related event types carry event specific information in additional attributes. These attributes are:

src, dst These attributes indicate the source and destination IP address of the sending or receiving peer respectively.

Valid for all event types.

srcid, dstid These attributes indicate the peer ID of the sending or receiving peer respectively.

The content of these attributes are encoded using the python functions repr and xml.saxutils.escape. The function repr returns a unique string representation of the input parameter, and the escape function returns an XML-escaped version of its input. Recall that the peer ID is a binary 20-byte value. The peer ID is first processed by the repr function to convert any non-printable to its python hexadecimal representation, i.e., the characters \x followed by the hexadecimal value. This string is then made into a valid XML attribute by the xml.saxutil.escape function, i.e., converting XML special characters such as & to &, with the exception of the quotation character, ", which is encoded using the python hexadecimal encoding (\x22). For a complete list of XML entity encodings see [75].

Valid for all event types except start_dl.

piece Denotes which piece a specific message refers to.

Valid for types piece, cancel, send, have and request.

begin Starting byte of a subpiece reference. Used together with the length parameter to denote a specific subpiece.

Valid for types piece, cancel, send and request.

length Number of content data bytes received or sent in a single piece message.

Valid for types piece, send and cancel.

down Denotes the number of downloaded and SHA1-verified bytes.

Only valid for type have.

nconns This attribute denotes the number of currently connected peers at the time of event ejection. This includes both locally and remotely initiated connections.

Valid for all event types.

port Indicates the TCP port of the remote peer.

Valid for type start_dl only.

direction Only valid for types have and bitfield. Used for differentiating between sent and received messages of these types. If the attribute is present and contains the value out, the message was sent by the measurement peer, otherwise it was received.

txtime The difference in time between the sending of the first subpiece of a piece and the reception of the last subpiece of the piece.

rxtime The difference in time between the first request of a piece and the reception of the last subpiece of the piece.

4.5.2 Common Log Formats

To facilitate the re-use of parsing software, it was decided as part of the measurement infrastructure, that parsed traces should be written to log files that adhere to a common format. This format is defined as follows:

- Fields are separated by spaces (ASCII code 32).
- Fields are defined as:
    1. The first field should always be the UNIX timestamp for the event.
    2. The second field should contain the message type, if any.
    3. Any following field may containg arbitrary information.

This is a simple and flexible logging scheme that allows us to use standard UNIX tools such as sed, awk and perl to parse the files without resorting to writing specialized parsers for each log file. In fact, the parsing software only assumes that the first field is a UNIX timestamp, and the rest of the fields are arbitrary. It is however recommended that the second field is a message type.

Figure 4.4 shows a portion of a log file generated from a BitTorrent application log.

```
1088079499.265901 piece
1088079499.267193 request 1311 196608
1088079499.269075 have 593
1088079499.275710 piece
1088079499.282697 have 6690
```

Figure 4.4: *Sample BitTorrent log file.*

Chapter 5

BitTorrent

The reported measurements have been done by having instances of the BitTorrent client software join several distribution swarms. An instrumented version of the reference BitTorrent client has been used to avoid potentially injecting non-standard protocol messages in the swarm. The client was instrumented to log all incoming and outgoing protocol messages together with a UNIX timestamp. The BitTorrent client is implemented in python, an interpreted programming language. The drawback with this is that the accuracy of the timestamps is reduced compared to the actual arrival times of the carrying IP datagrams. By comparing the actual timestamps of back-to-back messages at the application level with the corresponding TCP segments, we have found that the accuracy was approximately 10 ms.

Most of the traffic reported in this publication has been collected over a three week time period at two measurement points in Blekinge, Sweden. The first measurement point was the networking lab at BTH, Karlskrona, which is connected to the Internet through a 100 Mbps Ethernet network. The second measurement point was placed at a local ISP with a 5 Mbps link. Both measurement points were running the Gentoo Linux operating system, on standard PC hardware.

For the initial measurements, a number of twelve measurement have been done, each of them with a duration of two to seven days (Table 5.1). This first set of measurements were purely done with the instrumented client. An additional measurement with both application logging active and packet capturing running simultaneously has also been performed, for a total of thirteen measurements.

For the first measurement point, no significant amount of different software was running simultaneously with the BitTorrent client. At the second measurement point, the BitTorrent client was running as a normal application, together with other software such as web browsers and mail software. The first measurement point can be viewed as a dedicated BitTorrent client, while the second corresponds to normal desktop PC usage patterns.

5.1 Measurement details

Measurements 1 through 3 (Table 5.1) have been done with a single instance of the instrumented BitTorrent client running. As TCP is known to be very aggressive in using the network, this has been done to minimize the effects of several clients competing for the available bandwidth and to establish a point of reference for the rest of client sessions. Measurements 4 through 8 were started and done simultaneously, as were measurements 11 and 12. The other measurements were done with some temporal overlap, as shown in Figure 5.1.

Table 5.1: *Measurement summary.*

Number	Records	Start	Duration	Location
1	10770695	2004-05-03	2 days, 20 hours	BTH
2	10653466	2004-05-06	3 days, 19 hours	BTH
3	10990569	2004-05-12	4 days, 4 hours	BTH
4	12567283	2004-05-17	7 days	BTH
5	13691459	2004-05-17	7 days	BTH
6	11754838	2004-05-17	7 days	BTH
7	1943636	2004-05-17	7 days	BTH
8	7321166	2004-05-17	7 days	BTH
9[a]	687046	2004-05-13	3 days, 7 hours	ISP
10	2881803	2004-05-18	5 days, 23 hours	ISP
11	9252170	2004-05-22	7 days	ISP
12	5599997	2004-05-22	7 days	ISP
13	14803678	2004-06-26	7 days	BTH

[a]Unfortunately, the original data for this measurement was lost due to hardware failure. Thus, most analysis is not performed on this data, and only summary statistics are provided.

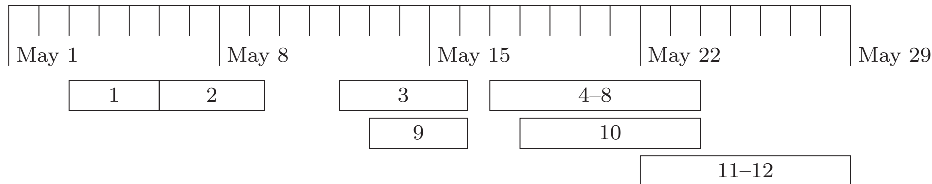


Figure 5.1: *Temporal structure of measurements.*

An important issue regarding traffic measurements in P2P networks is the copyright. The most popular content in these networks is usually copyrighted material. To circumvent this problem, we joined BitTorrent swarms distributing several popular Linux operating system distributions. Notably, we joined both the RedHat Fedora Core 2 (FC2) test and release versions. The FC2 'Tettnang' version was released on May 18th, while the rest of the content was available at the start of the measurements. This gave us a unique opportunity to study the dynamic nature of the FC2 swarms. The contents of the measured swarms are reported in Table 5.2. Two of the swarms have been measured from both measurement points to allow for comparisons, one with temporal overlap, and another without overlap.

5.2 Aggregate results

In this section we report some of the more salient results obtained from our measurements. We first summarise the download times and rates in table 5.3. It is observed that the time before our peer went into seeding mode varies from roughly 20 minutes up to 6.5 hours. As the content sizes vary with each measurement, we also provide the average download rate for the entire content, i.e., the size/time ratio. The download rates also show large disparity, with rates ranging from just over 129 kB to over 1.3 MB, with the three first measurements clearly being the most demanding

Table 5.2: *Content summary.*

Content	Pieces	Size	Measurement
RedHat Fedora Core 2 test3 CD Images	8465	2.2 GB	1–3
RedHat Fedora Core 2 test3 DVD Image	16708	4.3 GB	6, 10
Slackware Linux Install Disk 1	2501	650 MB	4
Slackware Linux Install Disk 2	2627	670 MB	5
Dynebolic Linux 1.3	2522	650 MB	7, 9
Knoppix Linux 3.4	2753	700 MB	8
RedHat Fedora Core 2 ‘Tettnang‘ CD Images	8719	2.2 GB	12,13
RedHat Fedora Core 2 ‘Tettnang‘ DVD Image	16673	4.3 GB	11

in terms of bandwidth utilization.

Table 5.3: *Download time and average download rate summary.*

#	Download	
	Time (s)	Rate (bps)
1	1930	1149520
2	1932	1147908
3	1681	1319445
4	2607	251424
5	3397	202644
6	23000	190416
7	1237	534282
8	6005	120153
9	2723	242776
10	23475	186570
11	19431	224927
12	9106	250989
13	2951	774420

A summary of session sizes and durations is reported in table 5.4. We also provide the number of sessions, unique peer IPs and peer client IDs.

Measurement 6 clearly stands out here, both with regards to mean session size and session length. Also, the maximum session size for this measurement is over twice that of any of the other measurements. The mean session size is also about twice that of the corresponding measurement of the same content (measurement 10). As measurements 6 and 10 have the top two session sizes, it is probable that the session size is related to the total content size (4.3 GB).

The minimum session lengths are all set to 0, indicating that they are all shorter than the accuracy provided for by the application logs. These very short sessions are also indicated in the minimum session sizes, and correspond to a session containing only a handshake or an interrupted handshake.

Another pertinent feature is the ratio of the number of unique IPs to the number of unique peers for measurement 8. The ratio for that measurement is slightly above 0.25, while none of the other

measurements are below 0.5. This might indicate either users stopping and restarting their clients several times, or users sharing IPs, such as peers subject to NAT.

Table 5.4: *Session and peer summary.*

#	Sessions	Session length (s)				Session size (MB)				Peers[a]	
		Mean	Max	Min	Std	Mean	Max	Min[b]	Std	ID	IP
1	29712	343	98991	0	2741	27.49	647.26	73	70.65	2024	1314
2	46022	233	117605	0	2316	27.15	646.03	73	64.05	1876	1394
3	28687	465	171074	0	3614	28.54	539.20	73	61.70	1913	1319
4	13493	750	143707	0	3942	49.88	671.99	73	100.65	1813	1143
5	12354	910	180298	0	4504	57.08	668.53	73	116.10	1747	962
6	10685	1207	223235	0	7016	74.25	3117.79	73	247.74	1033	619
7	4444	218	46478	0	1642	49.96	431.13	78	76.48	279	184
8	17287	231	87026	0	1972	33.11	695.94	73	109.31	1656	406
9	3043	294	29163	0	1719	21.62	408.05	78	42.27	193	166
10	9701	652	267497	0	5907	37.78	1499.85	73	109.08	444	305
11	43939	448	141509	0	3791	17.22	475.86	73	52.73	1841	1067
12	68288	197	292241	0	2580	8.31	987.89	73	30.63	2177	1152
13	52833	465	483996	0	4036	32.2	1652.83	73	99.4	3930	2440

[a] Unique peer client IDs and IP addresses
[b] This column measured in bytes.

Table 5.5 summarises the number of messages received on a per-message basis. In addition, column 5 shows the number of incoming connection requests collected in our measurements.

The *request* and *have* messages clearly dominate in terms of number of messages sent, while the *interested* and *not interested* messages are the least common. This is valid for all the measurements, except for measurement 2, which has almost 5 times more incoming *interested* messages than the measurement with the second highest number of *interested* messages.

The high number of *request* and *have* messages found in our measurements is expected, as the peer is acting as a seed for most of the time spent in the swarm. When seeding, a peer never receives *piece* messages, and downloading peer must request data by the *request* message, thus explaining their high number. The *have* messages are accounted for by the fact that every completed piece download results in such a message being transmitted.

Table 5.5: *Downstream protocol message summary.*

#	*request*	*not int.*	*piece*	*new conn.*	*bitfield*	*unchoke*	*have*	*int.*	*choke*	*cancel*
1	3316470	504	135615	29746	28024	27120	3651835	2905	26314	6500
2	3044768	489	135797	46047	45054	19117	3984881	14602	18061	9059
3	3276644	493	135682	28714	27092	40705	3941658	2430	39955	7628
4	5596270	406	40167	13502	12935	29628	1206000	2041	28640	14643
5	6163605	401	42176	12364	11827	32325	1197813	2059	31452	11508
6	4501907	191	277261	10688	9659	24239	2090892	2147	23639	6244
7	810019	52	40371	4445	4370	290	198885	230	122	1255
8	3347256	766	44328	17292	16623	9270	404038	2012	8579	18999
9	217336	37	40426	3045	2996	1114	139472	259	956	3061
10	838379	79	268429	9703	9181	13015	570367	692	11936	9085
11	1835910	470	268575	43957	42848	54090	4713440	2573	52458	17313
12	1118110	348	139943	68297	67373	37925	2619333	3242	36872	25047
13	8113100	711	139702	52865	50304	60524	6293438	9477	58925	24632

The summary of the outgoing messages in table 5.6 again shows the very low number of *interested* and *not interested* messages. The major bulk of the outgoing messages is however accounted for by

the *piece* messages. This is again an expected result, as *request* messages generate a *piece* message in response. The absence of transmitted *choke* messages for measurement 7 indicate that there has been a continuous exchange of data between peers. As for the *request* and *have* messages, these are tightly coupled to the number of pieces present in the content. The higher number of *request* messages is because these messages correspond to only a single subpiece.

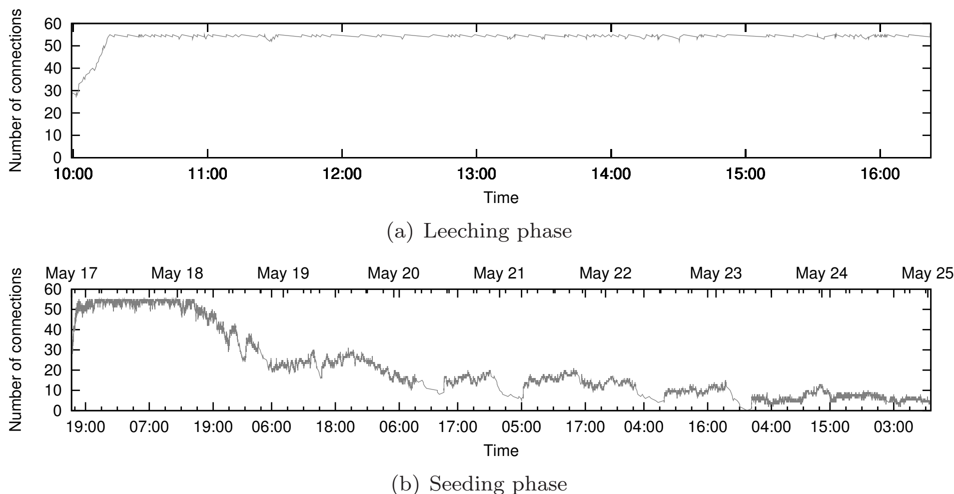
Table 5.6: *Upstream protocol message summary.*

#	*request*	*piece*	*not int.*	*unchoke*	*bitfield*	*int.*	*have*	*choke*	*cancel*
1	137007	3251948	63	11792	29714	68	8465	9553	970
2	137271	2964836	63	17471	46020	70	8465	13301	894
3	136738	3189175	62	16545	28682	64	8465	14085	1011
4	42709	5468908	76	25476	13489	86	2501	22740	855
5	44862	6032599	146	25759	12353	157	2627	23749	725
6	291200	4394389	91	23166	10661	197	16708	18943	555
7	40497	808844	18	4445	4444	18	2522	0	140
8	47413	3296616	100	19380	17281	136	2753	8672	423
9	40906	213693	16	3192	3042	19	2522	193	220
10	285650	753074	71	21304	9673	214	16708	15222	611
11	281921	1660868	67	35698	43927	157	16673	31279	812
12	145517	960802	76	49093	68271	125	8719	34570	701
13	141316	7940342	80	27332	52830	97	8719	23527	807

5.3 Swarm size dynamicity

After having downloaded all the data for a given torrent, the peer disconnects all connected seeds, and starts acting as a seed itself.

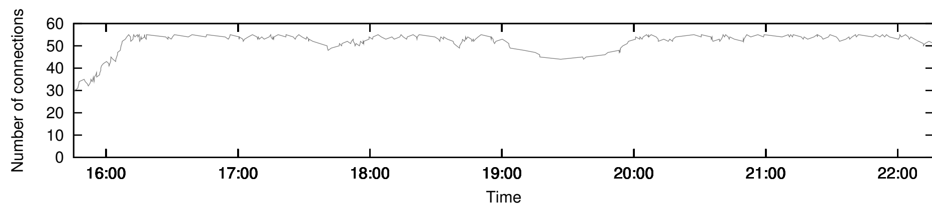
It is interesting to compare the seed phases of measurements 6, 10 and 11. The data in the first two was the test release of the RedHat Fedora Core 2 linux distribution, while the data in the last swarm was the final release of the same version. The final version was released on May 18. This event can be clearly seen at around 12:00 in Fig 5.2(b), at which time peers start disconnecting, most likely due to the new version of the distribution being released. The decrease in connected peers can also be seen in Fig 5.3(b).



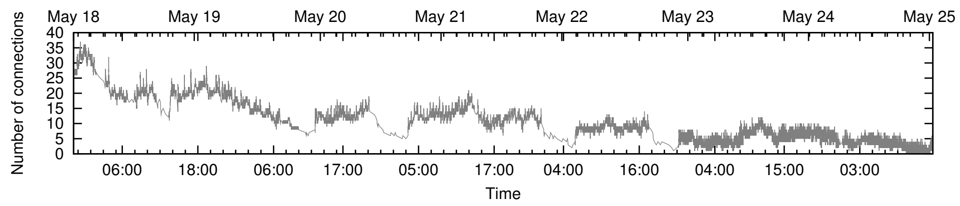
(a) Leeching phase

(b) Seeding phase

Figure 5.2: *Swarm size for measurement 6.*



(a) Leeching phase



(b) Seeding phase

Figure 5.3: *Swarm size for measurement 10.*

Chapter 6

Gnutella

Two sets of Gnutella measurements were performed at BTH. The first set was done in the early development phase of the measurement infrastructure (spring 2004), while the second set of measurements was performed one year later. During that year the Gnet has undergone major changes in terms of new protocol features. Functionality such as dynamic query, download mesh, PFS and UDP file transfers were adopted by a majority of servents, thus profoundly changing the characteristics of Gnutella network traffic. The research group at BTH has therefore decided to focus the analysis efforts on the more recent measurements since they were deemed to better reflect the current and future directions of Gnutella traffic characteristics. As a consequence, this report presents the recent set of measurements.

All results presented here were obtained from a 11-day long link-layer packet trace collected at BTH with the measurement infrastructure presented in Chapter 4.2. The Gnutella application flow reassembly was performed as described in Chapter 4.3.

The gtk-gnutella open source servent was configured to run as ultrapeer and to maintain 32–40 connections to other ultrapeers and 100 connections to leaf nodes. The number of connections is the vendor preconfigured value, which is close to the suggested values in [48, 47]. Although gtk-gnutella is capable of operation over UDP, this functionality was turned off. Consequently, the ultrapeer used only TCP for its traffic. No other applications, with the exception of an SSH daemon, were running on the ultrapeer for the duration of the measurements. One SSH connection was used to regularly check on the status of the measurements and the amount of free disk space. The SSH connection was idle for most of the time. The firewall was turned off during the measurements.

6.1 Session Statistics

A Gnutella session is defined here as the set of Gnutella messages exchanged over a TCP connection between two directly connected peers that have successfully completed the Gnutella handshake. The session lasts until the TCP connection is closed by either FIN or RST TCP segments. The session duration is computed as the time duration between the instant when the first handshake message (CLI_HSK) is recorded (at the link layer) until the measured time of the last Gnutella message on the same TCP connection. The session is not considered closed until both sides have sent FIN (or RST) segments.

An incoming session is defined as a session for which the CLI_HSK message was *received* by the ultrapeer at BTH. Outgoing sessions are sessions for which the CLI_HSK messages was *sent* by

the ultrapeer at BTH. Table 6.1 and Table 6.2 show the duration (in seconds), the number of exchanged messages and bytes for incoming and outgoing sessions, respectively. Table 6.3 show the same statistics when no distinction is made between incoming and outgoing sessions.

In the column denoted "Samples" the first number shows the number of valid Gnutella sessions that is used to compute the statistics. A Gnutella session is considered valid (in the sense that is used to compute session statistics) if the Gnutella handshake was completed successfully and at least one Gnutella message was transferred between the two hosts participating in the session. The number in paranthesis is the total number of observed sessions, valid and invalid. In this case, only 30 % of the all sessions were valid (31.5 % and 13.4 % when considering only incoming and outgoing sessions, respectively).

Table 6.1: *Incoming session statistics.*

Type	Max	Min	Mean	Median	Std	Samples
Duration (s)	767553 (8.9 days)	0.03	517.30	0.86	6780.99	173711 (551168)
Messages	7561532 (7.6M)	4	585.18	11	22580.99	173711 (551168)
Bytes	535336627 (535.3M)	780	53059	1356	2034418	173711 (551168)

Table 6.2: *Outgoing session statistics.*

Type	Max	Min	Mean	Median	Std	Samples
Duration (s)	470422 (5.4 days)	0.12	3949.86	2459.10	11170.80	7094 (52904)
Messages	2644660 (2.6M)	6	23145.15	15716.50	58627.75	7094 (52904)
Bytes	182279191 (182.3M)	1574	2173564	1457360	4458468	7094 (52904)

Table 6.3: *Incoming + Outgoing session statistics.*

Type	Max	Min	Mean	Median	Std	Samples
Duration (s)	767553 (8.9 days)	0.03	651.98	0.87	7036.85	180805 (604072)
Messages	7561532 (7.6M)	4	1470.34	11	25375.64	180805 (604072)
Bytes	535336627 (535.3M)	780	136258	1357	2219411	180805 (604072)

The tables show that outgoing sessions transfer about 40 times more data than incoming sessions. Furthermore, it appears that for incoming sessions, few sessions transfer the majority of data. This can be seen by comparing the mean and median values for messages and bytes. The explanation can be found by comparing the mean and median duration values for incoming sessions. It can observed that most incoming sessions have very short duration (< 1 second). Currently, no good reason could be found for this behavior. Most connections were terminated with a BYE message with code 200 `Node Bumped`.

6.2 Message Statistics

Table 6.4 displays the message size statistics for each Gnutella message type. The type UNKNOWN denotes messages with a valid Gnutella header, but with unrecognized message type. The messages are either experimental or corrupted. The type ALL is used for statistics computed over all messages, irrespective of type.

It can be observed that on average QUERY_HIT and QRP messages have the largest size. They are tightly followed by handshake messages, where the capability headers accounts for most of the data. It is interesting to notice that the maximum size of QUERY_HIT messages is 39 kB, which is an order of magnitude greater than the 4 kB specified by [46].

Table 6.4: *Message size statistics.*

Type	Max	Min	Mean	Median	Std	Samples
CLI_HSK	696	22	336.91	328	65.69	604072
SER_HSK	2835	23	386.83	369	145.69	597896
FIN_HSK	505	23	107.92	76	88.55	212162
PING	34	23	25.48	23	3.88	4151799
PONG	464	37	74.96	61	38.68	43727188
QUERY	376	26	70.17	55	46.40	129078986
QUERY_HIT	39161	58	590.28	358	1223.58	13242329
QRP	4124	29	608.60	540	596.70	1158596
HSEP	191	47	70.39	71	28.15	538834
PUSH	49	49	49.00	49	0.00	63040718
BYE	148	35	40.02	37	15.84	167726
VENDOR	177	31	36.45	33	19.51	10195389
UNKNOWN	43	23	23.53	23	3.24	38
ALL	39161	22	93.45	49	303.26	266715733

The message duration statistic can be useful to infer waiting times at application layer when a message is divided across two or more TCP segments. The statistic is defined as the time difference between the first and last TCP segments that were used to transport the message. When a message uses only one TCP segment the time duration for that message is zero.

Table 6.5: *Message duration statistics in seconds (100 μs resolution).*

Type	Max	Min	Mean	Median	Std	Samples
CLI_HSK	349.3015	0	0.0308	0	1.0412	604072
SER_HSK	52.2645	0	0.0032	0	0.1350	597896
FIN_HSK	68.6295	0	0.0057	0	0.2838	212162
PING	251.2914	0	0.0273	0	0.6309	4151799
PONG	2355.8650	0	0.0077	0	0.5881	43727188
QUERY	2355.8650	0	0.0035	0	1.3271	129078986
QUERY_HIT	480.8159	0	0.0243	0	1.0260	13242329
QRP	753.1904	0	0.1883	0	1.6019	1158596
HSEP	74.0482	0	0.0017	0	0.2186	538834
PUSH	135.5155	0	0.0023	0	0.2017	63040718
BYE	148.7292	0	0.0386	0	0.5194	167726
VENDOR	391.3439	0	0.0117	0	0.2451	10195389
UNKNOWN	1.0418	0	0.2995	0	0.4294	38
ALL	2355.8650	0	0.0065	0	0.9968	266715733

From the median column in Table 6.5 it can be observed that at least 50% of the messages require just one TCP segment. The PONG and QUERY_HIT message rows contain extreme values for maximum duration, 2355.9 seconds (approximately 39 minutes). These values are probably the result of malfunctioning or experimental Gnutella servents.

Table 6.6 shows interarrival times for messages received by the BTH ultrapeer and Table 6.7 shows interdeparture times for messages sent by the BTH ultrapeer.

Summing over the number of samples for each message type does not add up to the value shown in the number of samples for message type ALL. The reason is that the analysis software ignores messages that generate negative interarrival/interdeparture times. Such negative times appear because the application flow reassembly handles several (typically more than one hundred) connections at the same time. On each connection the timestamp for arriving packets is monotonically increasing. However, the interarrival/interdeparture statistics presented here are computed across

all connections. To ensure monotonically increasing timestamps even in this case, new mesages from arbitrary connections are stored in a buffer which is sorted by timestamp. The size of the buffer is limited to 500000 entries due to memory management issues. Table 6.20 shows that on average there are 280 incoming and outgoing messages. This means that the buffer can store about 30 minutes of average traffic and much less during traffic bursts. If there are messages that are delayed (see Table 6.5) due to TCP retransmissions or other events, they will reach the buffer too late and will be discarded.

Table 6.6: *Message interarrival time statistics (100 μs resolution).*

Type	Max	Min	Mean	Median	Std	Samples
CLI_HSK	28.4591	0.0001	1.7246	1.1256	1.8644	551148
SER_HSK	5185.0490	0.0001	19.6294	0.2090	92.1849	48432
FIN_HSK	1118.9920	0.0001	5.3165	2.1942	19.4800	178783
PING	13.5871	0.0001	0.2762	0.1931	0.2726	3457169
PONG	2.2624	0.0001	0.1404	0.0979	0.1383	9086918
QUERY	1.4514	0.0001	0.0343	0.0240	0.0340	59010007
QUERY_HIT	19.2778	0.0001	0.1842	0.0976	0.2661	6932327
QRP	50.0632	0.0001	2.0475	1.0534	2.8707	478451
HSEP	1780.4420	0.0003	6.1560	4.3834	8.4758	154742
PUSH	40.1396	0.0001	0.0677	0.0405	0.1157	24934450
BYE	1119.5930	0.0001	5.9160	2.3591	22.3494	160695
VENDOR	30.8037	0.0001	0.4346	0.2207	0.5993	9669915
UNKNOWN	51576.8600	3.0680	2075.3190	6.9379	9298.3600	35
ALL	9.8299	0.0001	0.02436	0.0169	0.0243	114663084

Table 6.7: *Message interdeparture time statistics (100 μs resolution).*

Type	Max	Min	Mean	Median	Std	Samples
CLI_HSK	5189.2340	0.0002	17.9655	0.1273	88.8506	52902
SER_HSK	28.4595	0.0003	1.7298	1.1287	1.8712	549456
FIN_HSK	5185.5150	0.0006	28.4784	0.3305	110.2372	33373
PING	20.5910	0.0001	1.3773	0.5077	2.1342	694550
PONG	2.7215	0.0001	0.1573	0.1012	0.1682	34639367
QUERY	12.1151	0.0001	0.0295	0.0003	0.0541	70066326
QUERY_HIT	19.2818	0.0001	0.2188	0.1285	0.2885	6309719
QRP	603.3599	0.0001	2.6350	0.0004	19.8572	680103
HSEP	358.3067	0.0001	2.5020	1.4089	5.8293	384084
PUSH	76.5303	0.0001	0.0429	0.0003	0.1713	38105019
BYE	3849.4550	0.0001	134.8121	77.2090	187.7784	7033
VENDOR	64.6689	0.0001	1.8253	1.1124	2.4838	525269
UNKNOWN	N/A	N/A	N/A	N/A	N/A	1
ALL	1.5450	0.0001	0.0178	0.0003	0.0353	152047214

The large interarrival and interdeparture times in handshake messages happen because once a servent reaches the preset amount of connections it will no longer accept or attempt new connections until one or more of the existing connections is closed. This behavior would also explain the large interarrival and interdeparture times for BYE messages.

6.3 Transfer Rate Statistics

This section present transfer rates in bytes/second and in messages/second for all Gnutella message types considered in this report. Each statistic is computed from 950568 samples, which is equivalent to approximately 11 days.

Table 6.8: *Handshake message rate statistics.*

Type	Dir	Max	Min	Mean	Median	Std
CLI_HSK	IN	12	0	0.58	0	0.79
CLI_HSK	OUT	30	0	0.06	0	0.56
SER_HSK	IN	20	0	0.05	0	0.48
SER_HSK	OUT	12	0	0.58	0	0.79
FIN_HSK	IN	9	0	0.19	0	0.46
FIN_HSK	OUT	18	0	0.04	0	0.34

Table 6.9: *Handshake byte rate statistics.*

Type	Dir	Max	Min	Mean	Median	Std
CLI_HSK	IN	4126	0	187	0	258
CLI_HSK	OUT	14519	0	27	0	273
SER_HSK	IN	12507	0	31	0	289
SER_HSK	OUT	4001	0	212	0	306
FIN_HSK	IN	982	0	15	0	42
FIN_HSK	OUT	4474	0	9	0	94

Table 6.10: *PING–PONG message rate statistics.*

Type	Dir	Max	Min	Mean	Median	Std
PING	IN	72	0	3.64	3	1.94
PING	OUT	17	0	0.73	0	1.56
PONG	IN	130	0	9.56	9	4.33
PONG	OUT	433	0	36.44	36	19.12

Table 6.11: *PING–PONG byte rate statistics.*

Type	Dir	Max	Min	Mean	Median	Std
PING	IN	1665	0	92	92	50
PING	OUT	503	0	19	0	45
PONG	IN	17043	0	1213	1173	541
PONG	OUT	26050	0	2235	2162	1179

Table 6.12: *QUERY–QUERY_HIT message rate statistics.*

Type	Dir	Max	Min	Mean	Median	Std
QUERY	IN	347	0	62.08	60	19.64
QUERY	OUT	875	0	73.71	69	34.08
QUERY_HIT	IN	531	0	7.29	5	9.82
QUERY_HIT	OUT	272	0	6.64	5	7.39

Table 6.13: *QUERY–QUERY_HIT byte rate statistics.*

Type	Dir	Max	Min	Mean	Median	Std
QUERY	IN	24101	0	4441	4317	1426
QUERY	OUT	46424	0	5088	4702	2511
QUERY_HIT	IN	1736791	0	4868	1912	23917
QUERY_HIT	OUT	360235	0	3355	1837	5229

Table 6.14: *QRP and HSEP message rate statistics.*

Type	Dir	Max	Min	Mean	Median	Std
QRP	IN	45	0	0.50	0	0.98
QRP	OUT	283	0	0.72	0	7.18
HSEP	IN	20	0	0.16	0	0.41
HSEP	OUT	23	0	0.40	0	0.68

Table 6.15: *QRP and HSEP byte rate statistics.*

Type	Dir	Max	Min	Mean	Median	Std
QRP	IN	47340	0	389	0	1408
QRP	OUT	152820	0	353	0	3660
HSEP	IN	940	0	8	0	21
HSEP	OUT	2185	0	32	0	58

Table 6.16: *PUSH and BYE message rate statistics.*

Type	Dir	Max	Min	Mean	Median	Std
PUSH	IN	1068	0	26.23	23	19.34
PUSH	OUT	4091	0	40.09	32	37.32
BYE	IN	40	0	0.17	0	0.43
BYE	OUT	118	0	0.01	0	0.15

Table 6.17: *PUSH and BYE byte rate statistics.*

Type	Dir	Max	Min	Mean	Median	Std
PUSH	IN	52332	0	1285	1127	948
PUSH	OUT	200459	0	1964	1568	1829
BYE	IN	1720	0	6	0	16
BYE	OUT	4956	0	1	0	11

Table 6.18: *VENDOR and UNKNOWN message rate statistics.*

Type	Dir	Max	Min	Mean	Median	Std
VENDOR	IN	6385	0	10.17	1	76.17
VENDOR	OUT	24	0	0.55	0	0.80
UNKNOWN	IN	1	0	0.00	0	0.01
UNKNOWN	OUT	1	0	0.00	0	0.00

Table 6.19: *VENDOR and UNKNOWN byte rate statistics.*

Type	Dir	Max	Min	Mean	Median	Std
VENDOR	IN	210702	0	347	33	2514
VENDOR	OUT	2197	0	44	0	81
UNKNOWN	IN	23	0	0	0	0.1
UNKNOWN	OUT	43	0	0	0	0.1

Table 6.20: *Gnutella (all type) message rate statistics.*

Dir	Max	Min	Mean	Median	Std
IN	6471	0	120.63	111	84
OUT	4164	0	159.96	153	61

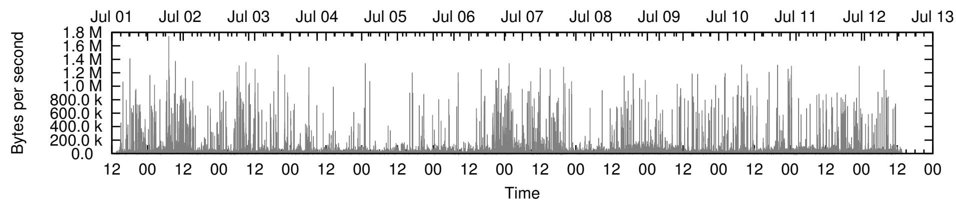
Table 6.21 shows the aggregate transfer rates for all messages types. Table 6.22 provides the summary statistics for the IP byte rates. It is interesting to note that the mean and median IP byte rates are very similar to the corresponding Gnutella byte rates shown in Table 6.21. These values alone would indicate that the compression of Gnutella messages does not yield large gains. However, if one takes into consideration the maximum and standard deviation values it can be observed that compression removes much of the burstiness from the application layer, leading to smoother traffic patterns. This effect can be also seen if one compares Figure 6.1 to Figure 6.2.

Table 6.21: *Gnutella (all type) byte rate statistics.*

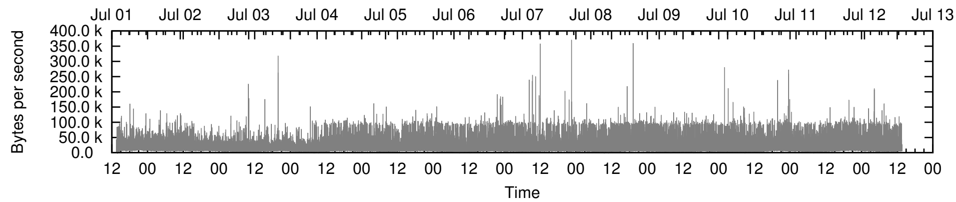
Dir	Max	Min	Mean	Median	Std
IN	1745341	0	12883	10113	24287
OUT	370825	0	13338	12062	7624

Table 6.22: *IP Byte rate statistics.*

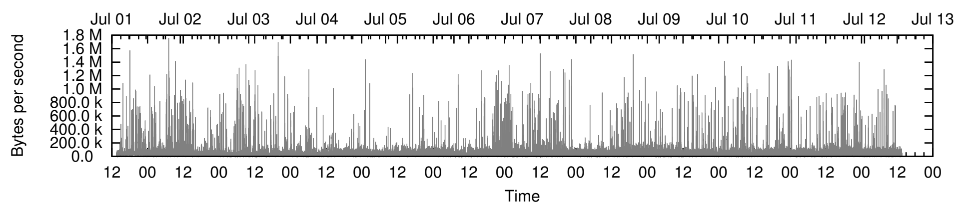
Dir	Max	Min	Mean	Median	Std
IN	249522	0	11536	10961	4075
OUT	176986	0	12668	12037	5722



(a) Incoming Gnutella Byte Rate

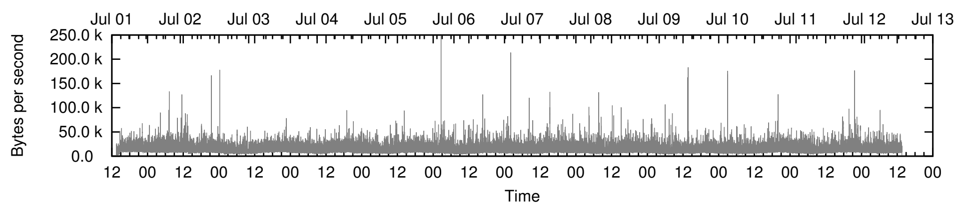


(b) Outgoing Gnutella Byte Rate

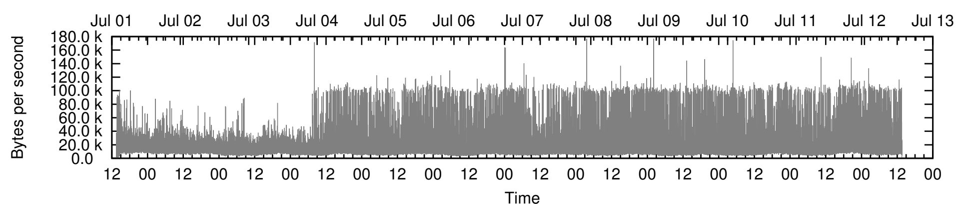


(c) Incoming + Outgoing Gnutella Byte Rate

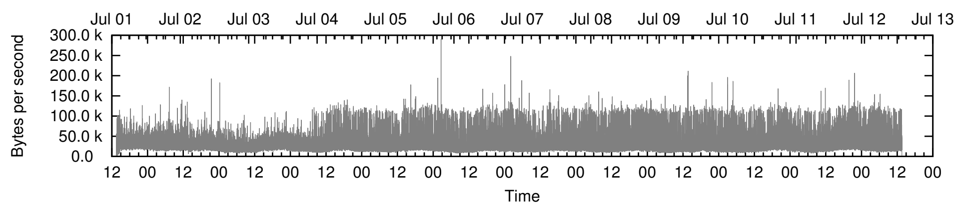
Figure 6.1: *Gnutella Transfer Rates.*



(a) Incoming IP Byte Rate



(b) Outgoing IP Byte Rate



(c) Incoming + Outgoing IP Byte Rate

Figure 6.2: *Gnutella Transfer Rates at IP layer.*

Appendix A

BitTorrent Application Log DTD

```
<!ELEMENT EVENTLIST (#PCDATA | EVENT)*>
<!ATTLIST EVENTLIST start_timestamp CDATA #IMPLIED>
<!ATTLIST EVENTLIST peertype CDATA #IMPLIED>
<!ATTLIST EVENTLIST version CDATA #IMPLIED>
<!ATTLIST EVENTLIST bound_ip CDATA #IMPLIED>
<!ATTLIST EVENTLIST bound_port CDATA #IMPLIED>
<!ATTLIST EVENTLIST tracker_ip CDATA #IMPLIED>
<!ATTLIST EVENTLIST tracker_port CDATA #IMPLIED>
<!ATTLIST EVENTLIST peer_id CDATA #IMPLIED>
<!ATTLIST EVENTLIST pieces CDATA #IMPLIED>
<!ATTLIST EVENTLIST piecesize CDATA #IMPLIED>
<!ATTLIST EVENTLIST nfiles CDATA #IMPLIED>
<!ATTLIST EVENTLIST totlen CDATA #IMPLIED>
<!ATTLIST EVENTLIST max_slice_length CDATA #IMPLIED>
<!ATTLIST EVENTLIST rarest_first_cutoff CDATA #IMPLIED>
<!ATTLIST EVENTLIST ip CDATA #IMPLIED>
<!ATTLIST EVENTLIST download_slice_size CDATA #IMPLIED>
<!ATTLIST EVENTLIST snub_time CDATA #IMPLIED>
<!ATTLIST EVENTLIST rerequest_interval CDATA #IMPLIED>
<!ATTLIST EVENTLIST max_uploads CDATA #IMPLIED>
<!ATTLIST EVENTLIST saveas CDATA #IMPLIED>
<!ATTLIST EVENTLIST min_uploads CDATA #IMPLIED>
<!ATTLIST EVENTLIST spew CDATA #IMPLIED>
<!ATTLIST EVENTLIST max_upload_rate CDATA #IMPLIED>
<!ATTLIST EVENTLIST minport CDATA #IMPLIED>
<!ATTLIST EVENTLIST http_timeout CDATA #IMPLIED>
<!ATTLIST EVENTLIST timeout_check_interval CDATA #IMPLIED>
<!ATTLIST EVENTLIST display_interval CDATA #IMPLIED>
<!ATTLIST EVENTLIST max_initiate CDATA #IMPLIED>
<!ATTLIST EVENTLIST max_message_length CDATA #IMPLIED>
<!ATTLIST EVENTLIST upload_rate_fudge CDATA #IMPLIED>
<!ATTLIST EVENTLIST check_hashes CDATA #IMPLIED>
<!ATTLIST EVENTLIST min_peers CDATA #IMPLIED>
<!ATTLIST EVENTLIST keepalive_interval CDATA #IMPLIED>
<!ATTLIST EVENTLIST maxport CDATA #IMPLIED>
<!ATTLIST EVENTLIST request_backlog CDATA #IMPLIED>
<!ATTLIST EVENTLIST bind CDATA #IMPLIED>
<!ATTLIST EVENTLIST max_rate_period CDATA #IMPLIED>
<!ATTLIST EVENTLIST url CDATA #IMPLIED>
<!ATTLIST EVENTLIST statfile CDATA #IMPLIED>
<!ATTLIST EVENTLIST report_hash_failures CDATA #IMPLIED>
<!ATTLIST EVENTLIST timeout CDATA #IMPLIED>
<!ATTLIST EVENTLIST responsefile CDATA #IMPLIED>
<!ATTLIST EVENTLIST max_allow_in CDATA #IMPLIED>
<!ELEMENT EVENT (#PCDATA)>
<!ATTLIST EVENT uploaded CDATA #IMPLIED>
<!ATTLIST EVENT downloaded CDATA #IMPLIED>
<!ATTLIST EVENT left CDATA #IMPLIED>
<!ATTLIST EVENT last CDATA #IMPLIED>
<!ATTLIST EVENT trackerid CDATA #IMPLIED>
<!ATTLIST EVENT event CDATA #IMPLIED>
<!ATTLIST EVENT numwant CDATA #IMPLIED>
<!ATTLIST EVENT port CDATA #IMPLIED>
<!ATTLIST EVENT txtime CDATA #IMPLIED>
<!ATTLIST EVENT rxtime CDATA #IMPLIED>
```

```
<!ATTLIST EVENT rxstart CDATA #IMPLIED>
<!ATTLIST EVENT direction CDATA #IMPLIED>
<!ATTLIST EVENT down CDATA #IMPLIED>
<!ATTLIST EVENT dst CDATA #IMPLIED>
<!ATTLIST EVENT dstid CDATA #IMPLIED>
<!ATTLIST EVENT nconns CDATA #IMPLIED>
<!ATTLIST EVENT type CDATA #IMPLIED>
<!ATTLIST EVENT timestamp CDATA #IMPLIED>
<!ATTLIST EVENT src CDATA #IMPLIED>
<!ATTLIST EVENT srcid CDATA #IMPLIED>
<!ATTLIST EVENT piece CDATA #IMPLIED>
<!ATTLIST EVENT begin CDATA #IMPLIED>
<!ATTLIST EVENT length CDATA #IMPLIED>
```

Appendix B

Acronyms

AD Anderson-Darling

AP Access Point

BTH Blekinge Institute of Technology

BSS Basic Structure Service

CCDF Complementary Cumulative Distribution Function

CDN Content Delivery Network

CGI Common Gateway Interface

CS Client-Server

CVM Cramér-von Mises

DHT Distributed Hash Table

DNS Domain Name System

DTD Document Type Definition

DOM Document Object Model

EDF Empirical Distribution Function

EPDF Experimental Probability Density Function

ESS Extended Service Set

F2F Firewall-to-Firewall

FTP File Transfer Protocol

GGEP Gnutella Generic Extension Protocol

GWC Gnutella Web Cache

HSEP Horizon Size Estimation Protocol

HTTP HyperText Transfer Protocol

HUGE Hash/URN Gnutella Extensions

HSEP Horizon Size Estimation Protocol

IID Independent and Identically Distributed

ISP Internet Service Provider

KS Kolmogorov-Smirnov

LN Leaf Node

LRD Long-Range Dependence

MLE Maximum Likelihood Estimation

ML Maximum-Likelihood

MPAA Motion Picture Association of America

NAT Network Address Translation

NFS Network Filesystem

NNTP Network News Transfer Protocol

OS Operating System

P2P Peer-to-Peer

PARQ Passive/Active Remote Queueing

PDF Probability Density Function

PFS Partial File Sharing

PIT Probability Integral Transform

PSTN Public Switched Telephone Network

QQ Quantile-Quantile

QoS Quality of Service

QRP Query Routing Protocol

RIAA Recording Industry Association of America

RMON Remote Monitoring

SAX Simple API for XML

SHA-1 Secure Hash Algorithm One

SMTP Simple Mail Transfer Protocol

SNMP Simple Network Management Protocol

SRD Short-Range Dependence

TCP Transport Control Protocol

TTL Time-To-Live

UDP User Datagram Protocol

UHC UDP Host Cache

UP Ultrapeer

URI Uniform Resource Indicator

URL Universal Resource Locator

URN Uniform Resource Names

UUCP Unix to Unix Copy Protocol

VoIP Voice over IP

WWW World-Wide Web

Bibliography

[1] Cachelogic. P2P in 2005.
http://cachelogic.com/research/p2p2005.php, 2005.

[2] Adrian Popescu. Routing on overlay networks: Developments and challenges. *IEEE Communications Magazine*, Vol. 43(8):22–23, August 2005.

[3] Wikipedia Encyclopedia. Peer-to-peer.
http://en.wikipedia.org/wiki/P2p, August 2005.

[4] Clip2. *The Annotated Gnutella Protocol Specification v0.4*. The Gnutella Developer Forum (GDF), 1.8th edition, July 2003. http://groups.yahoo.com/group/the_gdf/files/Development/.

[5] Jordan Ritter. Why Gnutella can't scale. No, really., February 2001.
http://www.darkridge.com-/~jpr5-/doc-/gnutella.html.

[6] Rüdiger Schollmeier. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In *Proceedings of the First International Conference on Peer-to-Peer Computing*. IEEE, 2001.

[7] Akamai.
http://www.akamai.com, August 2005.

[8] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, Volume 11(Number 1):17–32, February 2003.

[9] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM 2001*, pages 161–172, San Diego, CA, August 2001. ACM Press.

[10] Krishna P. Gummadi, Ramakrishna Gummadi, Steven D. Gribble, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. The impact of DHT routing geometry on resilience and proximity. In *Proceedings of the ACM SIGCOMM 2003*, pages 381–394, Karlsruhe, Germany, August 2003. ACM Press.

[11] Yatin Chawathe, Sylvia Ratnasamy, and Lee Breslau. Making gnutella-like P2P systems scalable. In *Proc. ACM SIGCOMM '03*, August 2003.

[12] Miguel Castro, Manuel Costa, and Antony Rowstron. Peer-to-peer overlays: structured, unstructured, or both? Technical report, Microsoft Research, Cambridge, UK, July 2004.

[13] Pablo Brenner. *A Technical Tutorial on the 802.11 Protocol*. BreezeCOM Wireless Communications, 1997.

[14] Xiaoyan Hong, Kaixin Xu, and Mario Gerla. Scalable routing protocols for mobile ad hoc networks. *IEEE Network*, pages 11–21, August 2002.

[15] ICQ.
http://www.icq.com, August 2005.

[16] Yahoo! messenger.
http://messenger.yahoo.com, August 2005.

[17] Msn messenger.
http://messenger.msn.com, August 2005.

[18] The SETI@Home Project. SETI@Home – the search for extraterrestrial intelligence.
http://setiathome.ssl.berkeley.edu/, February 2005.

[19] distributed.net. distributed.net.
http://distributed.net, February 2005.

[20] ZetaGrid. ZetaGrid.
http://www.zetagrid.net/, February 2005.

[21] Beowulf. http://www.beowulf.org, December 2005.

[22] Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.

[23] Ian Foster and Carl Kesselman. The Globus project: A status report. In *Proceedings of the Seventh Heterogeneous Computing Workshop*, pages 4–18, March 1998.

[24] David P. Anderson. BOINC: A system for public-resource computing and storage. In *Fifth IEEE/ACM International Workshop on Grid Computing*, pages 4–10, November 2004.

[25] Electricsheep. http://electricsheep.org, December 2005.

[26] Ian Foster and Adriana Iamnitchi. On death, taxes, and the convergence of peer-to-peer and grid computing. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, Berkeley, CA, USA, February 2003.

[27] Jonathan Ledlie, Jeff Schneidman, Margo Seltzer, and John Huth. Scooped, again. In *Proceedings of 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Berkeley, CA, USA, February 2003.

[28] Ian F. Akyildiz, Weilian Su, Yogesh Sankarusubramaniam, and Erdal Cayirici. A survey on sensor networks. In *IEEE Communication Magazine*, pages 102–114. August 2002.

[29] Deborah Estrin, Ramesh Govindan, John S. Heidemann, and Satish Kumar. Next century challenges: Scalable coordination in sensor networks. In *Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking*, pages 263–270, Seattle, WA, USA, August 1999. ACM.

[30] Chalermek Intanagonwiwat, Ramesh Govindan, Deborah Estrin, John Heidenmann, and Fabio Silva. Directed diffusion for wireless sensor networking. *IEEE/ACM Transactions on Networking*, 11(1):2–16, February 2003.

[31] Joanna Kulik, Wendi Heinzelman, and Balakrishnan Hari. Negotiation-based protocols for disseminating information in wireless sensor networks. *Wireless Networks*, 8(2/3):169–185, 2002.

[32] Bram Cohen. BitTorrent. http://www.bittorrent.com/, March 2006.

[33] BitTorrent specification.
http://wiki.theory.org/BitTorrentSpecification, February 2005.

[34] eDonkey.
http://www.edonkey.com, February 2005.

[35] NeoModus. DirectConnect.
http://www.neo-modus.com, February 2005.

[36] Sharman Networks. KaZaA.
http://www.kazaa.com, February 2005.

[37] National Institute of Standards and Technology. Specifications for secure hash standard.
http://www.itl.nist.gov/fipspubs/fip180-1.htm, April 1995. FIPS PUB 180-1.

[38] D. Eastlake 3rd and P. Jones. *US Secure Hash Algorithm 1 (SHA1)*, September 2001. RFC 3174.

[39] T. Berners-Lee, L. Masinter, and M. McCahill. *Uniform Resource Locators (URL)*, December 1994. RFC 1738.

[40] T. Berners-Lee, R. Fielding, and L. Masinter. *Uniform Resource Identifiers (URI): Generic Syntax*, August 1998. RFC 2396.

[41] Bram Cohen. BitTorrent protocol specification.
http://www.bitconjurer.org/BitTorrent/protocol.html, February 2005.

[42] Olaf van der Spek. BitTorrent udp-tracker protocol extension.
http://libtorrent.sourceforge.net/udp_tracker_protocol.html, February 2005.

[43] J.A. Pouwelse, P. Garbacki, D.H.J. Epema, and H.J. Sips. The BitTorrent P2P file-sharing system: Measurements and analysis. *4th International Workshop on Peer-to-Peer Systems (IPTPS'05)*, February 2005.

[44] Azureus.
http://azureus.sourceforge.net/, August 2005.

[45] J. Chapweske. Tree hash exchange format.
http://open-content.net/specs/draft-jchapweske-thex-02.html, February 2005.

[46] Tor Klingberg and Raphael Manfredi. *Gnutella 0.6.* The Gnutella Developer Forum (GDF), 200206-draft edition, June 2002.
http://groups.yahoo.com/group/the_gdf/files/Development/.

[47] Anurag Singla and Christopher Rohrs. *Ultrapeers: Another Step Towards Gnutella Scalability.* Lime Wire LLC, 1.0 edition, November 2002. http://groups.yahoo.com/group/the_gdf/files/Development/.

[48] Adam A. Fisk. *Gnutella Dynamic Query Protocol.* LimeWire LLC, 0.1 edition, May 2003. http://groups.yahoo.com/group/the_gdf/files/Proposals/Working Proposals/search/-Dynamic Querying.

[49] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of the Multimedia Computing and Networking (MMCN)*, January 2002.

[50] Gnutella protocol development.
http://www.the-gdf.org, December 2005.

[51] Raphael Manfredi. *Gnutella Traffic Compression*. The Gnutella Developer Forum (GDF), January 2003. http://groups.yahoo.com/group/the_gdf/files/Development/.

[52] Jean-loup Gailly and Mark Adler. zlib. http://www.gzip.org/zlib, August 2005.

[53] P. Leach, M. Mealling, and R. Salz. *A Universally Unique IDentifier (UUID) URN Namespace*, July 2005. RFC 4122.

[54] Jason Thomas. Gnutella generic extension protocol (GGEP). http://rfc-gnutella.sourceforge.net/src/GnutellaGenericExtensionProtocol.0.51.html, February 2002.

[55] Sumeet Thadani. Metadata extension. http://www.the-gdf.org/wiki/index.php?title=Metadata_Extension, 2001.

[56] Christopher Rohrs. *Query Routing for the Gnutella Network*. Lime Wire LLC, 1.0 edition, May 2002. http://groups.yahoo.com/group/the_gdf/files/Development/.

[57] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communication of the ACM*, Volume 13(Number 7):422–426, July 1970. ISSN:0001-0782.

[58] Adam A. Fisk. *Gnutella Ultrapeer Query Routing*. Lime Wire LLC, 0.1 edition, May 2003. http://groups.yahoo.com/group/the_gdf/files/Proposals/ Working Proposals/search/Ultrapeer QRP/.

[59] Thomas Schürger. *Horizon size estimation on the Gnutella network v0.2*, March 2004. http://www.menden.org/gnutella/hsep.html.

[60] G. Mohr. Hash/URN gnutella extensions (HUGE) v0.94. http://groups.yahoo.com/group/-the_gdf/files/Proposals/Working Proposals/HUGE/, April 2002.

[61] Raphael Manfredi. Passive/active remote queueing (parq). http://groups.yahoo.com/group/the_gdf/files/Proposals/Working Proposals/QUEUE/, May 2003. Version 1.0.a.

[62] Reliable udp (rudp) file transfer spec 1.0. http://groups.yahoo.com/group/the_gdf/files/Proposals/Pending Proposals/F2F/, February 2005.

[63] Pyda Srisuresh, Bryan Ford, and Dan Kegel. *State of Peer-to-Peer (P2P) communication across Network Address Translators (NATs)*. Internet Draft, October 2005. draft-srisuresh-behave-p2p-state-01.txt.

[64] Sam Berlin, Andrew Mickish, Julian Qian, and Sam Darwin. *Multicast in Gnutella. Document Revision 2*, November 2004. http://groups.yahoo.com/group/the_gdf/files/Proposals/Working Proposals/LAN Multicast/.

[65] Conny Palm. *Intensitätsschwankungen im Fernsprechverkehr*. PhD thesis, Royal Institute of Technology, 1943.

[66] Carey Williamson. Internet traffic measurement. 2001.

[67] "B. Claise". *"Cisco Systems NetFlow Services Export Version 9"*. Cisco Systems, October 2004. RFC3954.

[68] Van Jacobsen, C. Leres, and S. McCanne. Tcpdump. http://www.tcpdump.org, August 2005.

[69] Gerald Combs and contributors. Ethereal: A network protocol analyzer. http://www.ethereal.com, November 2005.

[70] Thomas H. Ptacek and Timothy N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, Inc., January 1998.

[71] Mark Handley and Vern Paxon. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proceeding of the 10th USENIX Security Symposium*, Washington, D.C., USA, August 2001. USENIX Association.

[72] Shawn Ostermann. Tcptrace. http://www.tcptrace.org, August 2005.

[73] Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated: The Implementation*, volume 2. Addison-Wesley, 1995. ISBN: 0-201-63354-X.

[74] Guido van Rossum et al. Python. Online at http://www.python.org, August 2005.

[75] W3C. *Extensible Markup Language (XML) 1.0*, 2004.