A Study of Peer-to-Peer Systems

JIA, Lu

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of
Master of Philosophy
in
Information Engineering

The Chinese University of Hong Kong
August 2009

Abstract of thesis entitled:
A Study of Peer-to-Peer Systems
Submitted by JIA, Lu
for the degree of Master of Philosophy
at The Chinese University of Hong Kong in June 2009

Peer-to-peer (P2P) systems have evolved rapidly and become immensely popular in Internet. Users in P2P systems can share resources with each other and in this way the server loading is reduced. P2P systems' good performance and scalability attract a lot of interest in the research community as well as in industry.

Yet, P2P systems are very complicated systems. Building a P2P system requires carefully and repeatedly thinking and examining architectural design issues. Instead of setting foot in all aspects of designing a P2P system, this thesis focuses on two things: analyzing reliability and performance of different tracker designs and studying a large-scale P2P file sharing system, Xunlei.

The "tracker" of a P2P system is used to lookup which peers hold (or partially hold) a given object. There are various designs for the tracker function, from a single-server tracker, to DHT-based (distributed hash table) serverless systems. In the first part of this thesis, we classify the different tracker designs, discuss the different considerations for these designs, and provide simple models to evaluate the reliability of these designs.

Xunlei is a new proprietary P2P file sharing protocol that has become very popular in China. Xunlei is interesting because it supports multiple protocols simultaneously - BitTorrent, eMule,

FTP and HTTP, and as well as a proprietary protocol that achieves very fast downloading speed. Its versatility expands the reach of its eco-system. Its speed makes one curious of its traffic engineering tricks. In the second part of this thesis, we study it by reverse-engineering: through a series of specially-designed experiments and careful dissecting of protocol messages, we discuss and speculate on the design of this popular protocol.

摘　要

近来，点对点系统（peer-to-peer systems）在互联网上得到了快速的发展并且获得了广泛的普及。点对点系统中的用户可以相互共享资源，这种方式减轻了传统服务器的工作负担。鉴于点对点系统的良好性能及可扩展性，很多专家学者以及工业界人士都对其产生了浓厚的兴趣。

然而，点对点系统是非常复杂的。建立一个点对点系统通常需要反复地仔细思考并检测系统设计的各个方面。在本文中我们并没有对点对点系统设计的每一个领域都有所涉及，而是将研究重点放在两个方面：分析点对点系统中不同"tracker"设计的性能，以及研究迅雷（Xunlei）这个典型的大规模点对点文件共享系统。

点对点系统中的tracker一般被用来查找拥有或部分拥有某一资源的节点（或用户）。从单一服务器的tracker到基于分布式哈希表（DHT）的tracker，点对点系统中的tracker设计可以采取很多种不同的形式。在本文的第一部分，我们对不同的tracker设计方法进行了归类，分析了这些方法所需考虑的实际系统问题，在此基础上提出了可以用来分析并且检验tracker设计性能的简单数学模型。

迅雷是一款在中国非常流行的私有的P2P文件共享协议。迅雷可以同时支持多种下载协议，如BitTorrent，eMule，FTP等等。在此基础之上，迅雷还采用了一款私有协议，可以获得极高的下载速度。同时，迅雷的多功能性扩大了其综合系统的用户覆盖率。迅雷的高速使得很多专家学者对于其工程设计的具体细节产生了浓厚的兴趣。在本文的第二部分，我们对迅雷进行了反向工程研究：通过一系列特别设计的实验和对私有协议消息类型的探究，我们对这一流行协议的具体设计细节进行了推测与讨论。

Acknowledgement

I would like to thank Professor Dah Ming Chiu, my supervisor, for his advice, help, encouragement, kindness and forgiveness throughout these two years. Prof.Chiu is such a nice, gentle and reputable professor that far beyond I deserve. It is him who has introduced and guided me into the world of P2P network research: inspiring my interests of study, teaching me hand-by-hand detailed mathematical techniques, helping me improve my English writing for papers over and over again...Looking back, from a fresh undergraduate student with few technical experience, to a graduating master who is getting ready for an overseas journey of research, I have learned so much from Prof.Chiu, not only in gaining abilities to do research, but also in holding an correct attitude towards research, and life. The time while I have Prof.Chiu as my supervisor, will definitely be a precious experience for my whole life.

I also appreciate very much the time and discussion I have had with my friends and colleagues in CUHK. They have inspired me many research ideas. Besides, with their accompany (working, spending leisure time together), I have spent very pleasant two years in Hong Kong.

Last but not least, thanks to my family for their incredible love, care and tolerance.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the past few years, peer-to-peer (P2P) systems have rapidly evolved and become an important part of the existing Internet culture. P2P systems' applications have covered many popular aspects of nowadays internet users' interests: file sharing, streaming, video-on-demand (VoD) and so on. To fulfill a P2P application, there are mainly three steps:

1. Finding target objects[1]
2. Finding swarm peers[2]
3. Exchanging resources[3] with other swarm peers

Specific designs of above three parts consist the design of a P2P system. There are usually several different methods to finish each step. To find the target object, we can use web search (as BitTorrent [20] does), DHT-based search (as eMule [12] does) or simply go through the resource lists provided by deployed servers (as PPLive [24] does). To find swarm peers, we can use deployed servers (such as BitTorrent's trackers), DHT-based

[1] can be a file in P2P file sharing application, a channel in P2P streaming application, or a video in P2P VoD application.

[2] A swarm is consisted by peers participating in the same session: downloading/uploading the same file, watching the same channel, or the same video. For convenience, we call peers in the same swarm to be swarm peers.

[3] can be the target object, or its information.

trackers (as that used in eMule) or simply gossip peer lists with other peers. To exchange resources with other swarm peers, incentive and download/upload policies are needed to decide peer and chunk selection strategies (like choke/unchoke and rarest-first policies used in BitTorrent), which help ensure a fair and efficient system performance.

A P2P system is a very complicated system. Its design considerations include, but definitely not restrict to above discussions. Instead of analyzing all aspects of P2P system designs, this thesis focuses on two things: analyzing reliability and performance of finding swarm peers (or the tracker function) in different P2P systems, and studying Xunlei, a popular multi-protocol P2P file sharing system, based on reverse engineering and measurement work.

In P2P systems, the term *tracker* originated from the design of the popular file sharing system BitTorrent [20]. A tracker is a server that is used to bootstrap a P2P system, an otherwise entirely distributed system. The most critical function provided by a tracker is to *introduce* other peers engaged in the same activity to a requesting peer. In order to perform this function, a tracker keeps track of peers as soon as they make a request. A tracker may also perform other related peer management functions. For example, peers may be required to periodically report to the tracker for keeping other statistics. Furthermore, a tracker can also be used to authenticate peers before providing them any service.

Subsequently, many other P2P content distribution systems, including many P2P streaming and Video-on-Demand (VoD) systems such as [24], adopted similar architectures as BitTorrent. All these P2P content distribution systems divide the content into many pieces and distribute them through different dynamically formed peer trees based on what pieces of content different peers are holding. This approach is referred to

as *data-driven*, or *unstructured* P2P content distribution algorithms. Its surprisingly good performance and adaptability in the face of peer churn and heterogeneous resource availability is attracting a lot of interest in the research community. This is particularly the case for the distributed algorithms for forming the peer overlay (peer selection algorithms in BT's terminology) and for scheduling the piece exchange among peers (piece selection algorithm in BT's terminology).

In the first part of this thesis, we focus on the design of the tracker function, which is a key enabler in this architecture. We use the word *tracker* to mean the service (provided by a BT tracker) rather than the server itself. There are two main approaches in tracker design, one is based on using deployed servers (we call server-based approach), and the other is based on using the peers themselves (we call peer-based approach). Based on these two approaches, there are many variants in the design, supporting scalability in the number of objects (e.g. files) and scalability in the number of peers simultaneously accessing the same object. For example, distributed hash table (DHT) [36, 30] and replication may be applied in tracker design. Our contribution is to provide a systematic description of the different designs of the tracker function, give a general discussion of the pros and cons of the different approaches, and present a simple model for DHT-based tracker design.

The chapter for this part is organized as follows. In section 1, we provide a taxonomy of different tracker designs and discuss the merit of different designs. In section 2, we present a simple model for DHT-based tracker design. In sections 3 we analyze its reliability by setting up simulations and comparing simulation and model results. We also derive some general observations from the model and the system parameters.

In the second part of this thesis, we study a popular multi-protocol file sharing system, Xunlei. According to Wikipedia [8],

Xunlei is a popular Chinese download manager and file sharing client that supports BitTorrent, eDonkey, Kad, and FTP. It is developed by Xunlei Networking Technologies, a Shenzhen (China) startup formerly known as Sandai Technologies. At the end of 2008, Xunlei has been used by 228 millions users and the amount of downloading has reached 5056TB per day [1].

Why has Xunlei attracted so many users? An example of user experience shown in Table. 1.1 gives us the likely answer: Xunlei has very high downloading speed.

Table 1.1: User experiences in file downloading (BT tasks, in Hong Kong)

No.	file size	client	↓ time	avg. ↓ speed	avg. ↑ speed
1	350MB	Xunlei(5.8.10)	10m58s	545.91KB/s	51.73KB/s
		BitTorrent(6.1.2)	59m47s	99.9KB/s	32.8KB/s
2	169.61MB	Xunlei(5.8.10)	24m51s	116.48KB/s	33.99KB/s
		BitTorrent(6.1.2)	45m04s	64KB/s	42.9KB/s
3	192MB	Xunlei(5.8.10)	22m44s	144.26KB/s	63.41KB/s
		BitTorrent(6.1.2)	16m44s	195.7KB/s	5.2KB/s
4	137.25MB	Xunlei(5.8.10)	14m09s	165.54KB/s	181.12KB/s
		BitTorrent(6.1.2)	>1h		
5	84.95MB	Xunlei(5.8.10)	16m29s	87.96KB/s	2.32KB/s
		BitTorrent(6.1.2)	>1h		
6	347MB	Xunlei(5.8.10)	23m56s	247.89KB/s	13.11KB/s
		BitTorrent(6.1.2)	40m00s	148.1KB/s	4.1KB/s
7	347.27MB	Xunlei(5.8.10)	23m29s	252.38KB/s	47.44KB/s
		BitTorrent(6.1.2)	52m27s	112.9KB/s	19.5KB/s
8	175MB	Xunlei(5.8.10)	27m04s	110.45KB/s	3.28KB/s
		BitTorrent(6.1.2)	9m29s	315.2KB/s	2.7KB/s
9	349MB	Xunlei(5.8.10)	18m34s	321.54KB/s	64.54KB/s
		BitTorrent(6.1.2)	30m52s	193KB/s	1.0KB/s
10	350.79MB	Xunlei(5.8.10)	18m05s	331.07KB/s	7.74KB/s
		BitTorrent(6.1.2)	59m47s	99.9KB/s	32.8KB/s

Besides the well-known P2P file downloading product, Xunlei also provides its users *Xunlei portal* for a variety of information source, plus *Gougou search* for searching video, music and other entertainment resources, *Xunlei Kankan* for P2P VoD application, *Xunlei youyou* for updating and downloading games and *Web Xunlei* [16, 13, 9]. It is interesting to note that 1) most of these products are based on Xunlei's P2P file downloading technology and 2) they are not isolated from each other. These products work together and form the *Xunlei Ecosystem.*

Instead of studying the whole Xunlei ecosystem, we focus only on its P2P file downloading application. Our study is based on reverse engineering and measurement work, which consist of several steps. Mainly we want to understand the following questions:

1) *What is Xunlei*? 2) *Being a multi-protocol system for P2P file downloading, how does a Xunlei client inter-operate with swarms that are speaking different languages (protocols)*? 3) *In Xunlei network, what functions do its servers provide*? 4) *Why does Xunlei achieve high downloading speed? Does it adopt any private (non BitTorrent or eMule compatible) protocols? How do they work*?

We answer the first question in section 1 of this chapter by giving an overview of Xunlei and its key components. Then we focus on Xunlei's multi-protocol strategy for file downloading and we explain its detailed process in section 2, where we show, through a number of experiments, that how a Xunlei client participates in BitTorrent or eMule network, gets peer information in swarms and exchanges resources with them (which answers the second question), from which we notice that besides those popular open-source P2P protocols, Xunlei also takes advantages of UDP's light-weight and flexibility: it designs a data transmission algorithm and frequently uses it during the file downloading process. For our third step, we study Xunlei's

UDP-based private protocol, including but not restricting to its message types, data structures, error control and congestion control mechanisms. We further analyze the reasons for Xunlei's good performance based on a number of measurement studies. We believe our results will be a useful resource for the understanding and analysis of Xunlei and other P2P designs.

All of our experiments about Xunlei are performed via Wireshark [15] and the results are derived from packet level analysis.

This thesis is organized as follows. In chapter 2 we introduce the background study of P2P tracker designs and previous work related to Xunlei. In chapter 3 we give an analysis of different tracker designs in P2P systems. In chapter 4 we present our results about the study of Xunlei and the conclusion is given in chapter 5.

□ **End of chapter.**

Chapter 2

Background Study

The tracker function, mostly finding swarm peers, is a key component of P2P system design. The inherent "distributed" property of P2P networks requires their users to find more swarm peers more efficiently to gain a better system performance. When BitTorrent protocol was first brought forward, the concept of "tracker" also emerged. At that time, a tracker was referred as a deployed server who kept a record of peers in the same swarm and replied a subset to requesting peers. The way to implement a tracker server and the messages used for peers and trackers to communicate with each other were well defined in BitTorrent's original design [21].

Later, BitTorrent, as well as many other P2P clients, enabled DHT to fulfill tracker function in a distributed way. Again, the way to implement a DHT tracker and the message types were introduced and well defined in BitTorrent design [29].

Although the technology of implementing trackers (both server-based and DHT-based trackers) is quite mature, few researchers focus on analyzing why implementing trackers in these ways, not mention to summarize or classify different tracker designs and design considerations.

The server-based tracker adopts traditional client-server mode. Both the research theories and implementation considerations related to it have been well-studied for many years. DHT-based

system (including DHT-based tracker), however, is a new coming technology whose concept was only brought forward a few years ago.

In the early days when DHT was first introduced (from 2002 to around 2004), researchers paid more attention to design and improve DHT algorithms. During that time, many different DHT algorithms were brought forward and compared [36, 30, 33, 2, 42]. Different design parameters were analyzed to improve system performance (latency, lookup ratio, bandwidth usage .etc) [28, 23].

Later after that, the population of P2P systems increased so quickly that the scalability became the most important requirement for system design. Researchers and system designers began to utilize DHT in real P2P applications (file sharing, video-on-demand .etc). Generally speaking there are three levels for utilizing DHT in P2P systems, namely finding swarm peers, publising/searching files and storing file chunks, among which finding swarm peers is the most wildly-used DHT application, although there do exist some system designers who would like to explore more about DHT and utilize it for file publishing/searching, or even file/chunk storage [40].

How about the performance of DHT-based systems? Recently DHT modeling also attracts many scholars' attention. A number of models about DHT performance have been brought forward and most of them are based on reliability theory, with considerations of peer churn in P2P systems [19, 39, 38, 27]. Nowadays DHT systems are mainly used for P2P applications, so DHT modeling can be classified into two parts.

- Modeling peer churn.

Being a basic property of P2P systems, peer churn is the foundation for DHT modeling. Most of DHT modeling papers share some similarities in this part. They model the peer churn

as a renewal process with lifetime (or residual lifetime) and repair time (death time) distributions. Lifetime is mostly heavy-tailed distributed (i.e. Pareto), as observed in real P2P systems by [18]. But exponential distribution is also used for startup as well as simplicity, which turns out to be a performance lower bound [27]. There are also some measurement work that focus on identifying peer behaviors in P2P networks [35, 37, 18].

- Modeling DHT behaviors.

Isolation probability/time and query success ratio are typical metrics for DHT performance modeling. Due to the importance of successor, authors in [39] study DHT's successor isolation probability, especially that in Chord. According to Chord's algorithm to keep successors, they model it as a node keeping a constant number of neighbors and define the isolation as a node losing all its neighbors.

For query success ratio, Guang Tan et al. provide a simple model for it in [38]. Upon deriving residual lifetime, they provide a straightforward formula for query success ratio, under the assumption that the path length remains unchanged even when a certain intermediate node fails.

Finding swarm peers is only one part of a P2P system design, yet one of the most important parts. Besides analyzing different tracker designs, we also perform a reverse engineering and measurement study of Xunlei, a multi-protocol P2P file sharing system. This study can help us know more about P2P system design details.

Xunlei is a proprietary P2P file downloading system targeting at Chinese users. Although it is very popular (has attracted over 200 million users) and achieves good performance (high file downloading speed), it has not been much studied and understood. The only paper about Xunlei [41] is a short paper that briefly discusses the general functions of Xunlei servers and some

preliminary measurement work.

The methodology of our work about Xunlei is inspired by the reverse engineering study of Skype [34]. Through analysis of traces of Skype's messages, it was able to deduce many aspects of Skype's system design. Gu et al [22] introduce and study a special designed UDP-based data transfer mechanism with a series of analysis and measurement work about TCP and UDP, which influenced our thinking about Xunlei UDP flows' influence to other flows, and to the network. [24] provides a measurement study of a large-scale P2P VoD system, covering many aspects of architectural design issues, which is a useful reference for our measurement work about Xunlei.

□ **End of chapter.**

Chapter 3

Analysis of P2P Tracker Designs

1 Tracker design in P2P systems

1.1 A taxonomy of tracker designs

For the data-driven P2P content distribution architecture, it is necessary for each peer to discover other peers engaged in the same content distribution session, as well as what pieces of content these peers have. The tracker usually only supports the discovery of peers. The discovery of what pieces peers hold is normally accomplished by *gossip* [20], in other words, by a peer directly querying its neighboring peers. What pieces peers hold changes frequently with time, so in most scenarios only gossip can provide the most timely information without overburdening a server and incurring excessive network overheads. While it is possible, broadly speaking, to consider this (providing piece information) also part of the tracker function, we take a narrower view. That is, the tracker only maps an object (distribution activity) to a set of peers (partially) holding this object.

Therefore, the tracker needs to deal with only two kinds of information: (a) objects, and (b) peers; and provide the mapping between them. Objects are the files (in P2P file sharing or VoD)

or video channels (in P2P streaming). A tracker should be able to serve multiple objects simultaneously. Peers are the users downloading the objects. Each peer registers with the tracker for the object it is downloading; and requests for a set of other peers downloading the same object. Although a broader view of the tracker can include additional interactions between the tracker and the peers (e.g. statistics collection), we assume the minimum responsibility for the tracker in this study. Tracker design can be classified by the following three dimensions:

Who provide the tracker function? There are basically two choices: using *deployed servers* (**DS**), or using peers (**P**). In the latter case, it is possible to rely on only a subset of the more powerful peers known as supernodes.

How are objects assigned to tracker nodes? In the same P2P system, there may be many objects made available for sharing. Instead of having one tracker node serving all these objects, multiple tracker nodes (whether DS or peers) can share the load. The assignment can be by manual configuration (**M**), or via a distributed hash table (**DHT**).

How are peers assigned to tracker nodes? A large number of peers may be accessing the same object simultaneously, causing too much load for a single tracker node to handle the load. There may be other locality and reliability reasons for having multiple tracker nodes serve a single object. In this case, the assignment depends on whether the tracker nodes are deployed servers or peers. In the former case, the assignment can be based on user choice, if tracker nodes are explicitly advertised to users (**U**), or can be automatic (**A**), if the tracker node must be found by a DHT mechanism. In the latter case, the assignment has to be automatic (**A**).

Let us now consider some examples of these different designs below.

In the classic BitTorrent, the tracker is a server and the bind-

ing of the tracker node to the object is advertised in a meta-file (the "torrent" file) [20]. A user (peer) can choose the tracker based on which meta-file it selects to use (or a specific tracker in a meta-file with multiple trackers). The peer then contacts the tracker to find other peers downloading the same object. We can refer to this design as (DS+M+U).

Another popular file sharing system, eMule, uses DHT to let the tracker function be shared by peers themselves. It uses a particular DHT algorithm known as Kademlia [30, 19]. The basic idea of any DHT algorithm is that it provides a mapping from an object name to a *target node*[1] that keeps some information about the object of interest. In reaching this node, the lookup process may have to traverse several intermediate nodes. A well-designed DHT also provides some redundancy (via replication) in the paths reaching any object. The mapping from the object to the set of trackers for the object is then stored at the target node. In our taxonomy, this design can be labeled (P+DHT+A).

A third example is the PPLive VoD system. According to the designers [24], the tracker function is provided by several deployed servers, and a DHT is used to allocate the objects (video files) to this set of servers. This design can be labeled as (DS+DHT+A).

It is interesting to note that it is not uncommon for a P2P system to simultaneously rely on two different mechanisms to support the tracker function, with one of the mechanisms used as a back-up. For example, some versions of BitTorrent also use (P+DHT+A) as a back-up; whereas eMule is also able to use a (DS+M+U)[2] in parallel. In the PPLive VoD system, a more recent version also includes a DHT to avoid *server filtering*, a simple technique to disable a P2P service.

[1]Here we do not differentiate the term *node* and *peer*. We use them iteratively throughout this thesis.

[2]Known as ED2K.

1.2 Design considerations

In designing the tracker function, there are many considerations. Many of them are not quantifiable. We discuss them briefly here.

Ease of implementation: A simple client server model should be simpler than DHT, and this can be the reason for the original tracker design.

Legal liability or management responsibility: There may be legal liability in running a tracker. It also incurs management chores. So a serverless (based on DHT) design is very desirable.

Costs: There are also some costs associated with running a tracker, e.g. the server and bandwidth costs. With a serverless tracker, these costs are absorbed by the peers.

Flexibility: Implementing tracker in servers certainly gains more control for the content distributor (in the case when content comes from distributor rather than from the peers themselves). For example, the content provider may make peers in different networks/countries use different trackers and form different sessions.

Security: On the one hand, server-based tracker can be subjected to DoS attacks; or the access to the tracker can be easily filtered out (for example by an ISP who wants to disable the P2P system). On the other hand, server-based tracker can be used to implement some access control policies.

While the above considerations are all important and could decide the tracker design, another important consideration is the reliability which directly affects user perception. This metric can be quantitatively evaluated, by a system model presented in the next section.

2 A reliability model for DHT-based tracker design

2.1 DHT basics

The basic principle of DHT (distributed hash table) is mapping each object and node into the identifier space $\{0, 1, ..., 2^k - 1\}$[3] using constant hashing. For each node or object, its identifier is unique and is also the only resource to identify itself. To avoid confusion, we call the object's identifier to be the *key* and the node's identifier to be the *ID*. With the key, an object can be mapped to their *responsible node* that is in charge of this object and its information. Basically an object can be assigned to an node with the ID closest to the key of this object. Methods to define "closest" are different from protocols. Chord defines "closest" node as the first node whose ID is larger than the key in the clockwise direction along Chord ring[36]. Kademlia defines it as the node with the smallest distance based on XOR metric[30].

Two basic functions performed by any DHT algorithm are *put(key)* and *get(key)*. Given the key, we can find the responsible nodes and assign an object into DHT network with the function *put(key)* and later we can retrieve it by finding the responsible nodes with *get(key)*. It is easy to see that finding the responsible nodes is a basic process in both *put(key)* and *get(key)*. This process is named *lookup* in DHT, where lookup messages are forwarded from the original requesting node to the responsible node hop by hop. One of DHT's important achievements is that, it exponentially decreases the *lookup* length by each hop while keeping the ID of next-hop intermediate node closer and closer to the key of the object. DHT table is an application-layered routing table that contains information of a subset of nodes in

[3] k is usually a sufficiently large value that can make sure no identifiers conflict among different nodes or objects.

the network, including node ID, its IP address and some other information which can be used for advanced features.

To handle churn, which is a typical peer behavior in P2P networks, DHT adopts certain mechanisms. *Stabilization* is a method widely used in many DHT algorithms. It refreshes DHT table entries according to a constant or randomized period, which allows DHT to detect and to replace failed nodes, as well as to note and to add new-joined nodes. Some DHT algorithms also adopt more casual and flexible ways to do refreshment. Kademlia[30] refreshes the information via *refresh upon lookup*. As its name implies, Kademlia refreshes its DHT table entries by each received lookup message.

2.2 Model preliminaries and assumptions

As discussed in previous subsections, DHT-based tracker design uses *lookup* message to perform tracker function and we define the success of lookup process as, given the key, eventually finding the responsible peer. A responsible peer is in charge of a particular range in DHT identifier space. The mechanism used to decide the responsible range depends on specific protocols. For example, Chord defines the responsible range of a peer as the ID range between its proceeding node (the nearest counter-clockwise node in the Chord ring) and itself.

For a particular lookup message, it traverses a path of intermediate peers, where each peer has several other peers serving as backups or *redundancies*. Here the need for redundancy in each hop is due to the dynamic behavior of P2P systems. Peer's join and departure are happening every time when users start or terminate a P2P application. Each peer (with a unique ID) first enters into the system, starts a number of P2P sessions and then stays in the system for some time. During this time, this peer is called to be *alive* and correspondingly the period it spent or

it is alive in the system is called peer's *lifetime*. When sessions are finished, this peer will leave the system with or without notification. Peers can also leave the system in the middle of a session voluntarily, or due to some uncontrollable reasons, such as link failure.

The most significant influence introduced into P2P systems by peer churn is the stale information, which affects system performance in many ways. Without further detection and repair, stale information may direct lookup messages to failed intermediate peer(s) which causes lookup failure. Another influence introduced by peer churn is the dynamic system population. Experiments in [24] show that system population changes with the time of a day. But when the system has been evolved for a long time and if we look into a particular time slot, like around 8:00 pm, system population stays around a particular level without large fluctuation. This is the steady state where rates of peer's join and of peer's departure are almost the same. Throughout this thesis, we only consider this steady state. To simplify analysis, we further assume a constant system population, which is achieved by the implementation that once a peer fails, it rejoins the system immediately as a brand new peer with a new ID.

To handle problems caused by peer churn, most DHT algorithms introduce refreshment mechanisms. Each time when DHT performs a refreshment, peer's liveness is verified and failed peers are replaced. Here we assume that DHT's refreshment mechanism is *stabilization* with a constant interval, for the reason that periodic stabilization is the most wildly used refreshment method in DHT systems (as discussed in last subsection). Stabilization is a process performed by each peer individually and no central control or central clock is used here. Besides of these, we further assume that each peer starts a number of lookups for randomly chosen keys during their lifetimes, and the lookup interval is exponentially distributed. Hence, for a

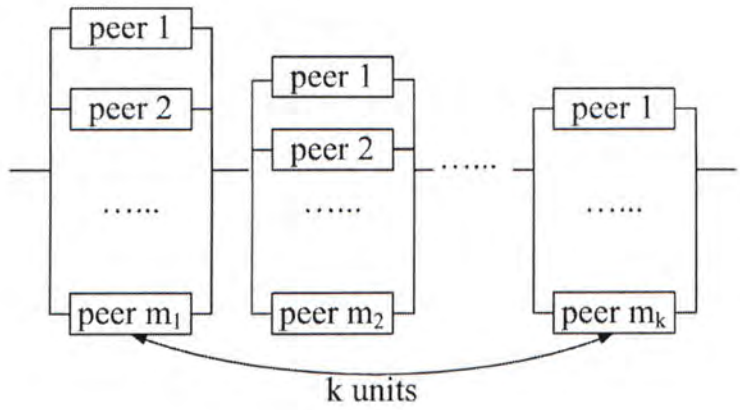


Figure 2.1: System structure of DHT-based tracker design

particular peer, lookup arrival is a poisson process.

From above analysis we may see that the operation of a real DHT system is very complicated. Analyzing the system by considering all the details is neither realistic nor necessary. Hence instead of giving a complicated model capturing all the aspects of the system, we set up a simple model that captures the basic features of DHT-based tracker design, which is discussed in following subsections.

2.3 Model description

As shown in Fig. 2.1, we model DHT-based tracker design as a *serial* set of units where each unit is composed of a *parallel* set of sub-units. The serial units are lookup hops and parallel sub-unites are redundant peers for each hop in DHT table entries. The number of unites is the hop length for a lookup and the number of sub-unites in each unit is the number of redundancies in each lookup hop. We measure system reliability by a simple criterion: *lookup success ratio*, which is determined by the system structure and single peer (sub-units in Fig. 2.1) reliability.

Single peer reliability

We consider single peer reliability to be the probability that the corresponding peer is alive when a lookup message arrives. Obviously, single peer reliability depends on peer's liveness and peer's liveness depends on its remaining lifetime, or formally *residual lifetime.* Originally, peers's lifetime L is determined by lifetime distribution $F(x)$. When DHT performs stabilization, peer's liveness is verified and its remaining lifetime will be changed. Since stabilization is an individual behavior for each peer, it is not possible for us to know when it happened previously. Hence we assume it to be uniformly distributed within each peer's lifetime. Therefore peer's remaining lifetime after stabilization can be derived based on following residual lifetime theorem[32]:

Lemma 2.1 *Let $F(x)$ be the CDF of peer's lifetime L, then the CDF of peer's residual lifetime is given by:*

$$F_R(x) = P(R < x) = \frac{1}{E[L]} \int_0^x (1 - F(z))dz \qquad (2.1)$$

Real world experiment[18] shows that P2P user's lifetime has long-tailed behavior and it is approximately pareto distributed. Together with exponential distribution's memoryless property, we use pareto and exponential (for comparison) as lifetime distributions to validate our model and simulation results throughout this thesis. And this analysis is based on following lemma:

Lemma 2.2 *The CDF of residuals for Exponential lifetimes with $F(x) = 1 - e^{-\lambda x}$ is given by:*

$$F_R(x) = P(R < x) = 1 - e^{-\lambda x} \qquad (2.2)$$

and the CDF of residuals for Pareto lifetimes with $F(x) = 1 - (\frac{\beta+x}{\beta})^{-\alpha}, \alpha > 1$ is given by:

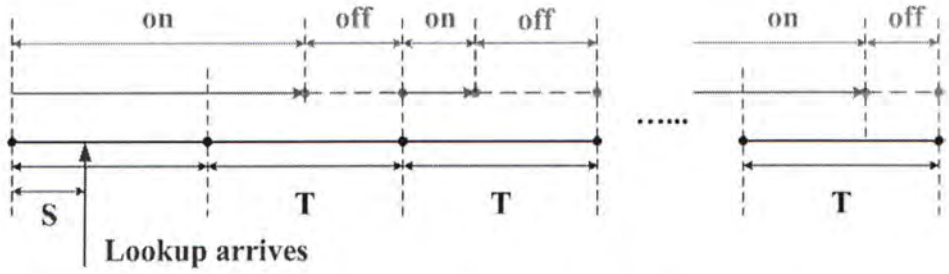


Figure 2.2: Peer's on-off status and lookup arrival

$$F_R(x) = P(R < x) = 1 - (\frac{\beta + x}{\beta})^{-(\alpha-1)} \tag{2.3}$$

Lemma 2 is not a surprising outcome. Under memoryless exponential distribution, peer's lifetime is refreshed after stabilization. For "long-tailed" Pareto distribution, residuals can gain a longer tail, which means that the mean value of residual lifetime is longer than that of the original lifetime. This is consistent with the fact that in most P2P systems, users who survive in the system for some time are likely to remain on-line for longer period than the new arriving users.

In DHT-based P2P systems each participating peer is in charge of a particular range of DHT structure and we call this peer *responsible peer*. For a particular key, the responsibility can be re-allocated from one peer to another (due to peer's churn) according to DHT's stabilization mechanism. If the original responsible peer is failed, DHT's stabilization can detect this failure and replace it with a new one. Hence from each key's point of view, the corresponding responsible peer possess two status: *on* and *off*. The time it stays in the status *on* after stabilization is determined by its residual lifetime. After this peer fails, the status switches to *off* until next stabilization replaces it with a new responsible peer and switches the status back to *on*. Fig. 2.2 shows the *on-off* process of a particular responsible peer. Similar on-off process also happens to intermediate peers

for a particular lookup message.

Upon deriving the on-off model, we examine lookup arriving process within a certain stabilization interval, which is shown in Fig. 2.2. S is the time interval between lookup's arrival and last stabilization. Since we previously assumed a memoryless exponential distributed lookup interval, it is refreshed after stabilization and S is exponentially distributed. Therefore for a peer to be alive when a lookup arrives, its residual lifetime need to be longer than S.

Above analysis provides the basis for following proposition:

Proposition 2.3 *Let $F_R(x)$ be the CDF of peer's residual lifetime R and T be the constant stabilization interval. With an exponential(λ') distributed lookup arrival interval and alternate on-off status, single peer reliability p is given by:*

$$\begin{aligned} p &= Pr\{R \geq S\} = \int_0^\infty Pr\{R \geq x\} \times f_S(x)dx \\ &= \int_0^T (1 - F_R(x)) \times \frac{\lambda' e^{-\lambda' x}}{\int_0^T \lambda' e^{-\lambda' y} dy} dx \end{aligned} \tag{2.4}$$

For exponential and pareto distributed lifetime, single peer reliability is given in following corollary:

Corollary 2.4 *Assume peer's lifetime is exponential(λ) distributed, then single peer reliability is given by:*

$$p = \frac{\lambda'(1 - e^{-(\lambda+\lambda')T})}{(\lambda + \lambda')(1 - e^{-\lambda' T})} \tag{2.5}$$

and for Pareto(α, β) lifetime distribution, single peer reliability is given by:

$$
\begin{aligned}
p &= \frac{\beta^{\alpha-1}\lambda'}{1-e^{-\lambda' T}} \int_0^T (x+\beta)^{-(\alpha-1)} e^{-\lambda' x} dx \\
&= \frac{\beta^{\alpha-1}\lambda' e^{\lambda'\beta}}{1-e^{-\lambda' T}} \sum_{n=0}^{\infty} \frac{(-\lambda')^n}{n!} \frac{(T+\beta)^{n-\alpha+2} - \beta^{n-\alpha+2}}{n-\alpha+2}
\end{aligned} \tag{2.6}
$$

System reliability

As shown in Fig. 2.1, we model the DHT-based tracker design as a series of units and each unit is composed by some sub-unites. Let k represent for the number of the serial units, namely the number of hops for each DHT lookup. Let m_1, m_2,...m_k represent for the number of sub-units, namely the number of available serving peers in each hop. Based on reliability theory, we can calculate the system reliability as shown in following proposition.

Proposition 2.5 *For a DHT-based P2P system, the probability that a particular lookup will be successful is given by:*

$$
\mathcal{R}_{DHT} = \prod_{i=1}^{k} (1 - (1-p)^{m_i}) \tag{2.7}
$$

and the system reliability is given by:

$$
E[\mathcal{R}_{DHT}] = E[\prod_{i=1}^{k} (1 - (1-p)^{m_i})] \tag{2.8}
$$

where k is the lookup length and m_1, m_2,...m_k is the number of available serving peers in each hop, and p is single peer reliability given by Proposition 2.3.

Now the key questions for completing the model become:

- How many hops do we have for each lookup?

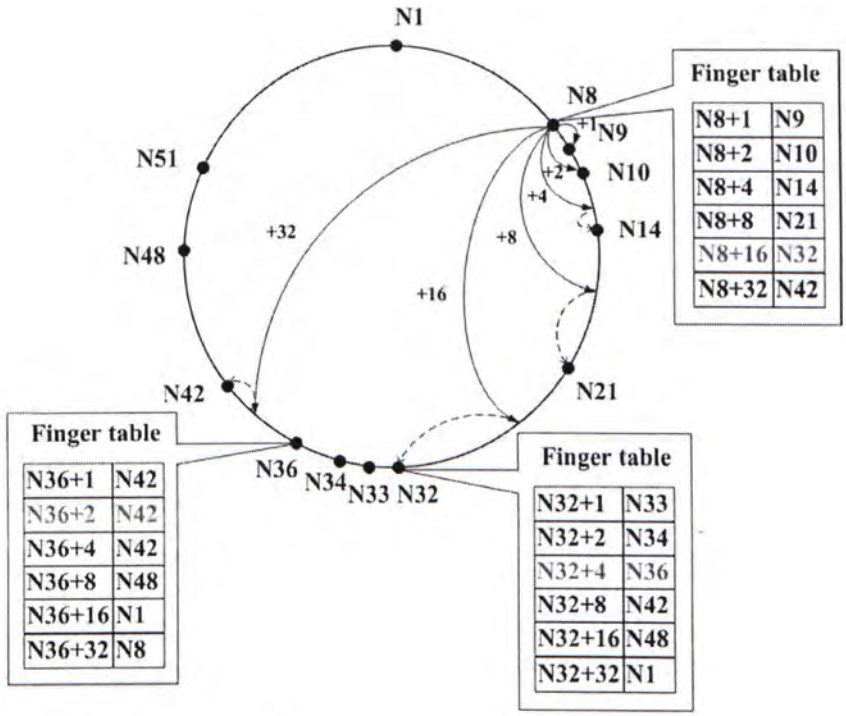


Figure 2.3: Chord finger table example.

- How many redundancies do we have for each hop?

We use Chord [36] for example to answer above questions, since the design of Chord shows DHT's basic principles without introducing too many advanced features [36].

To answer the first question, we assume that nodes are uniformly distributed among the ID space[4] and lookup keys are randomly chosen from the key space. Since lookup lengths in DHT-based P2P systems are bounded by $\log_2 N$, where N is system population, and the lookup keys are randomly chosen, we further assume that the mean value of lookup length is $(\log_2 N)/2$.

To answer the second question, Fig. 2.3 shows the basic structure of a Chord ring and the DHT tables of node N8, N32 and N36. Suppose that at the beginning N8 starts a lookup for key 38. According to Chord's algorithm, the lookup message will be

[4]This is also a property of constant hashing.

forwarded from N8 to N32, N36 and N42 hop by hop. Knowing that N42 is the responsible node for key 38, this lookup is finished successfully at N42.

Problems happen when one of the above intermediate nodes is not alive. Suppose N8 finds that N32 is unreachable when it tries to forward the lookup message to N32. Then N8 needs the redundancies in this hop to continue the lookup. According to [36], alternative nodes, or the redundancies for a hop, can be easily found in the table entries preceding that of the failed node. So in our example, when N32 fails, N8 still has N9, N10, N14 and N21 as alternative nodes, which are 4 redundancies for this hop.

From above example, we may have two observations about redundancy: 1)the number of redundancies in each hop depends on the ID distance between the intermediate node and the key; 2)the number of redundancy decreases by each hop. These observations can be intuitively understood based on DHT's basic principle: forwarding the lookup messages closer and closer to the target key in each hop.

Recall Proposition. 2.5 that the system reliability is the mean value of a stochastic function related to m_1, m_2,...m_k, k and single peer reliability p. Nevertheless, instead of analyzing the stochastic behavior of the system, we focus on system's mean performance in this model. So instead of calculating the mean value of a complicated stochastic function, we simplify this by approximating the mean of a function to be equal to the function of the mean. More precisely, we approximate that:

$$\begin{aligned} E[\mathcal{R}_{DHT}] &= E[\mathcal{F}(k, m_1, m_2...m_k, p)] \\ &\approx \mathcal{F}(E[k], E[m_1], E[m_2]...E[m_k], p) \end{aligned}$$

and this approximation will be evaluated later by simulations.

To complete the model, we need to decide the mean values of m_1, m_2,...m_k and k. Still using Chord for example, it contains

$\log_2 N$ entries in each routing table and since lookup keys are randomly chosen among identifier space, we further assume that on average the number of available severing peers in the node who starts the lookup, namely $E[m_1]$, is $\log_2 N/2$. It is interesting to note that, $E[m_1]$ is equal to the average path length $E[k]$ (both are equal to $(\log_2 N)/2$, as discussed previously). Together with the fact that the number of redundancies decreases by each hop, we are trigged to assume that on average the number of redundancies decreases by one in each hop. With these assumptions and analysis, the system reliability of a Chord-like DHT system can be derived as shown in following corollary:

Corollary 2.6 *For a Chord-like DHT system with a constant population N in steady state, the system reliability can be approximated to:*

$$E[\mathcal{R}_{DHT}] \approx \prod_{k=1}^{\frac{\log_2 N}{2}} (1-(1-p)^k) \tag{2.9}$$

where p is single peer reliability given by Proposition 2.3.

In next section, we'll set up simulations to do more analysis zhout system reliability. The numerical results of Corollary 2.6 will also be calculated and compared with simulation results in next section.

3 Reliability analysis

3.1 Related parameters

From Proposition 2.5, the only criterion we used to measure reliability of DHT systems, namely *lookup success ratio*, is related to system population and single peer reliability. From Theorem 2.3, this single peer reliability further depends on peer's lifetime, system population, stabilization interval and lookup rate.

Therefore, for the reliability of DHT-based tracker design we have four related parameters, which are discussed respectively as follows:

Peer's lifetime is a determinant parameter for churn rate in P2P systems. Peer's churn may produce stale information and consequentially deteriorate system's performance. The churn rate, somehow decides to what degree it can deteriorate system performance. For system designers, they always prefer peers in their systems to stay as long as possible. Unfortunately, the inherent properties of dynamic and self-organization for P2P systems make it almost impossible for designers to control peer's churn behavior. So instead, some system designers choose to deploy more stable *super nodes* to weaken the influence of peer churn. Besides the mean value of peer's lifetime, its distribution is also an influence. With the same mean value, different distributions may have different effects on system performance.

System population directly reflects system's popularity. While designers are glad to observe a large population in their systems, they also need to handle the problems caused by it, such as the workload, especially when there are certain bottlenecks in the system. As discussed in previous section, system population is dynamic due to peer's join and departure. A larger peer join rate can increase system population while a larger peer departure rate can shrink it. With the same join and departure rate, systems will stay at the steady state with population keeping at a certain level without large fluctuation.

Stabilization interval determines the rate for DHT to do stabilization. Intuitively the system can gain a better performance with a smaller stabilization interval. Methods for stabilization usually depend on system implementations. Right until here, we always assume a constant stabilization in our model.

Lookup rate shows the popularity for a particular object (e.g. a file) and all together determine system's workload. For

each alive peer in the system, it performs several lookups with randomized lookup interval. Throughout this paper we assume this interval to be exponentially distributed.

3.2 Simulation setup

In this subsection, we analyze the system reliability by setting up simulation experiments with real implemented DHT algorithms. These experiments can be used to evaluate the related parameters discussed in last subsection, as well as our theory model.

Simulation methodology

About P2P simulations and simulators, methodology and tools have always been discussed and questioned[31]. Generally speaking, there are three main methodologies for simulating P2P networks: 1)packet-level; 2)overlay-level and 3)model-level.

Due to the popularity of some packet-level simulators like NS-2[10] and OPNET[14], some scholars take advantage of their well-organized and update-in-progress documents and use them as P2P simulators with or without modifications[43, 26]. Packet-level information helps simulations to resemble real world applications very well, but it also aggravates simulator's workload at the same time. Hence it is almost impossible to simulate large-scaled P2P networks with packet-level simulators.

Model-level simulation is always the favorite choice for many researchers due to the simpleness to write some codes that simulate model principles. And since they only simulate the models (or the algorithms), they can always get good results showing the similarity between their model and simulation results. But model-based simulation's lack of considerations of real network details also makes any claims on results hard to be validated and further to be believed with confidence by other researchers.

With so many disadvantages of above two kinds of simulators, overlay-level simulator is hence a good choice for simulating P2P networks. [31] have done a survey about existing overlay-level simulators for P2P networks, and 9 of them are well-studied and compared. Among them, we choose P2PSIM[11, 28], a discrete-event simulator for structured P2P networks, for the reason that it implements the original designs of many DHT algorithms and it also adopts the KING topology[5] to model the lower-layer network performance. Moreover, it has already included peer churn pattern. Among all the DHT algorithms implemented in P2PSIM, we choose Chord as the DHT prototype for simulation model, for the reason that Chord grabs all basic principles of DHT without introducing too many advanced features.

Comparison of theory and simulation model

In both theory or simulation models, operations of DHT-based system are decided by three main modules, namely peer behavior, protocol behavior and network behavior.

Peer behavior: In this part, theory and simulation models adopt exactly the same scene: Peer's ID is uniformly distributed among ID space. Each peer spends some time in the system and during this time it starts several lookups for randomly chosen keys, with exponential distributed lookup intervals. And peer's life time is exponential(λ) or Pareto(α,β) distributed. At a certain time it crashes, then re-joins the network immediately as a new node with a new ID. This behavior also ensures a constant population for the system.

Protocol behavior: DHT protocol behaviors include several parts: initial state, dealing with lookup, handling node join/failure and periodic stabilization. We do not consider the initial state either in our theory model or in the simulation, for

[5]KING topology is a pairwise latency matrix derived from measuring the inter-node latencies of 1024 DNS servers using the KING method.

the reason that instead of the highly dynamic initial state, we concern more about DHT's steady state behavior.

For dealing with lookup, in both models lookup messages are forwarded from one peer to another and finally to the one who is responsible for it. There are several redundancies in each hops and these redundancies come from DHT table entries. The difference between theory and simulation models is that, in theory model we try to model the overall average performance while the simulation is consisted by many single lookup cases.

In both models node failure is handled by constant stabilization with the same interval. In theory model we have assumed that after each stabilization, peer's life is refreshed immediately. But in simulation, under the real implementation of DHT, it takes a short period from the beginning of stabilization until peers are refreshed, which can be seen as a quick transit status.

When a new node joins, in simulation it performs DHT's *join* function. With this operation, take Chord[36] for example, it is possible to induce: 1) correct successor pointer and finger table; 2) correct successor pointer but incorrect finger table; 3) incorrect successor pointer. Among all these three cases, only the last one will yield incorrect lookup and the probability for this is quite small. Hence we omitted this influence of new node join process in theory model.

Network behavior: Although P2PSIM is an overlay-based simulator, it adopts KING topology to model network's RTT latency. So in simulation model, it is possible for a lookup to fail under a very large RTT latency[6]. But in theory model we assume that as long as a peer is alive, its neighbors can always reach it.

[6]if the RTT latency is larger than the timeout

3.3 Results

In this subsection, we present model and simulation results. We examine system reliability under different peer's lifetimes, different system populations, different stabilization intervals and different lookup rates (four related parameters as discussed above). Each result is derived by averaging five rounds of simulations and each simulation runs for six hours of simulated time. The ranges and default values of parameters used in model and simulation are summarized in Table 3.1.

Table 3.1: Parameters' range and default value

Parameters	Range	Default value
Peers' average life time	10–60 minutes	30 minutes
System population	64–8192	1024
Average DHT stabilization interval	1.5–10 minutes	1.5 minutes
Average lookup rate	1–10 times per life	30 minutes

The way we choose the range for different parameters is based on the considerations of real world situation. In our theory and simulation models, peer's average lifetime is ranging from 10 minutes to 1 hour. This is reasonable since it is only the mean value. For different distributions, i.e. exponential or pareto distribution, we may have some peers with lifetimes much longer (or much shorter) than this mean value. And the lookup rate is ranging from 1 time per life to 10 times per life, which is also reasonable since 10 times per peer life is a frequency large enough for real P2P applications. Stabilization interval is set to be from 1.5 to 10 minutes. The reason why we didn't evaluate results with a very large stabilization interval is that, for DHT algorithms that don't refresh DHT tables by lookup messages, stabilization interval can not be set too large. Otherwise the system may be merged with too many stale information to work normally. System population is set to be from 64 to 8192. 8192

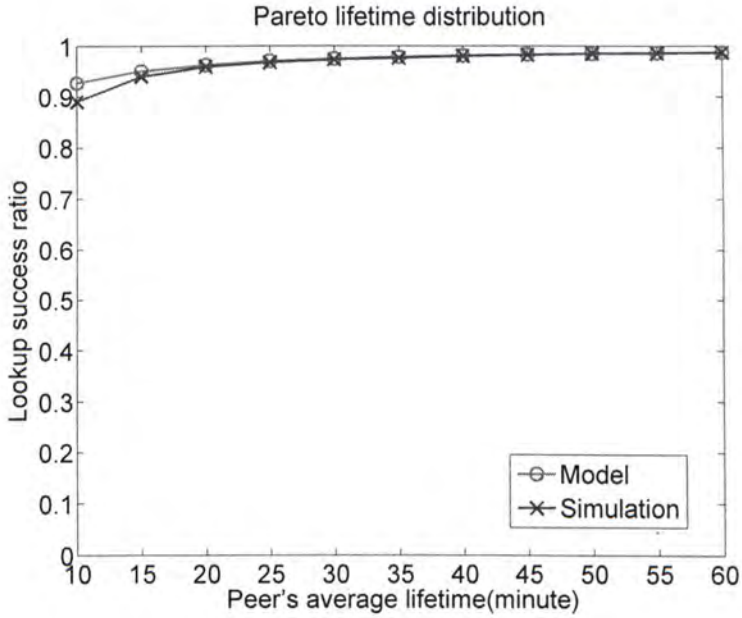


Figure 3.4: Influence of peer's lifetime

is an on-line population large enough for P2P systems and an larger population introduces too much workload that may slow down or even crash the simulator.

We show and discuss the theory model and simulation results respectively in following subsections.

Influence of peer's lifetime

Consistent with our intuition, as peer's average life time increases, lookup success ratio increases in both model and simulation results (see Fig. 3.4). It is interesting to note that, although we haven't performed model-level simulation, we still get good results here. Model and simulation results match each other quite well, with only negligible differences. Two observations can be gained from these results:

- Model results are always slightly better than simulation results.

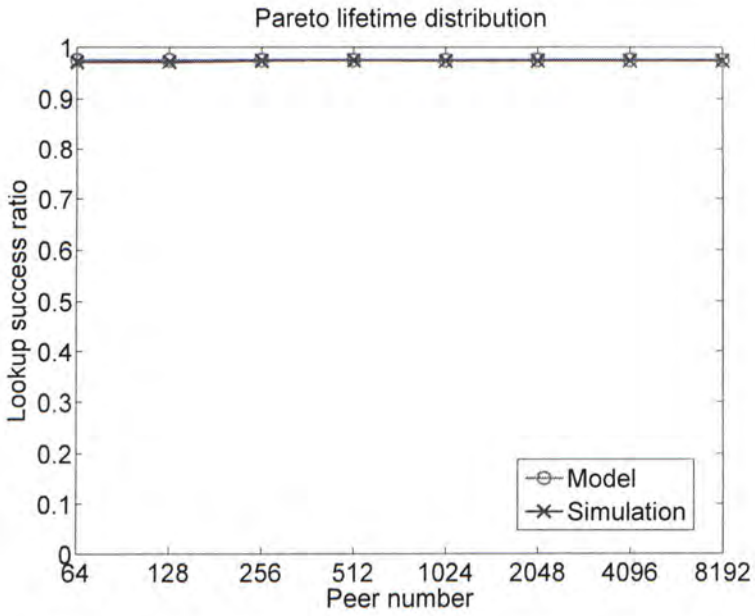


Figure 3.5: Influence of system population

We believe this is due to the fact that we have omitted stabilization's transit status and lower-layer network influence (RTT) in our theory model (as described previously). And both of these two cases may deteriorate the lookup success ratio to some extend.

- The difference between model and simulation results increases with a higher churn rate.

We believe this is due to the fact that we didn't consider the new node join influence in our theory model. Although as discussed before, with a very low probability that a new node join may fail a lookup, when the churn rate is really high, the highly frequent new node join does affect the lookup success ratio. Hence, as shown in Fig. 3.4, the stabler these peers are, the better model results resemble simulation results.

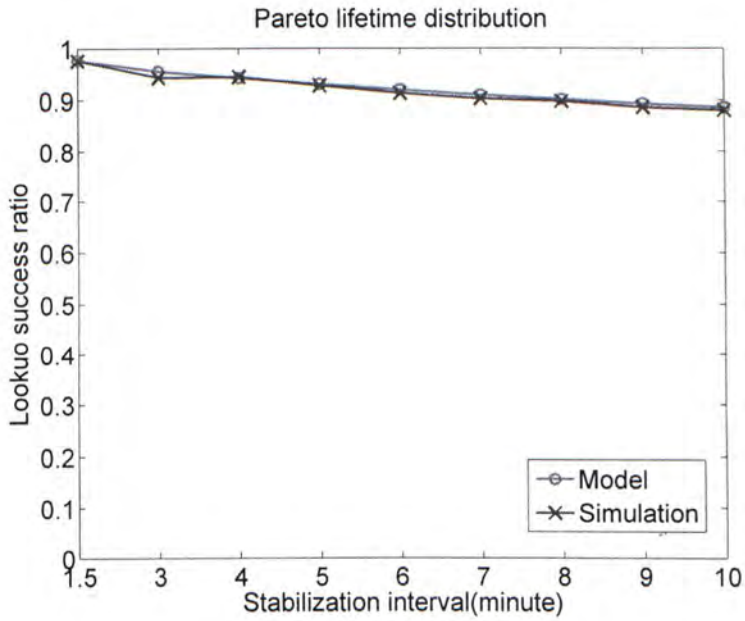


Figure 3.6: Influence of stabilization interval

Influence of system population

In this subsection we examine the influence of system population. Results are shown in Fig. 3.5.

From Fig. 3.5 we may see that lookup success ratio almost keeps at a stable level with different system populations, which shows the scalability of DHT-based system. Theoretically, this scalability is gained from the fact that DHT distributes workload to every participating peer. Hence, although workload increases with system population, so does the number of peers that can share the burden.

Influence of stabilization interval

To analyze the influence of stabilization, we change its interval from 1.5 minutes to 10 minutes, while keeping other parameters as constants. Both model and simulation results are shown in Fig. 3.6.

From Fig. 3.6, it is easy to see that lookup success rate de-

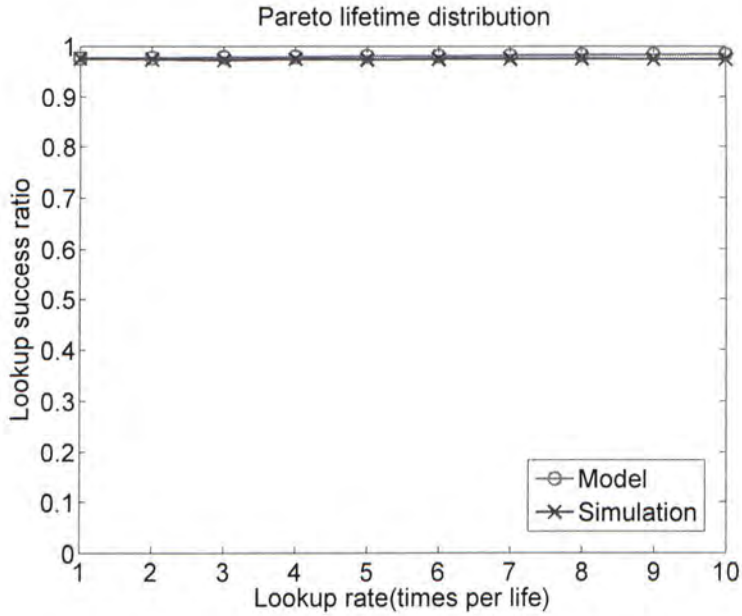


Figure 3.7: Influence of lookup rate

creases as stabilization interval increases in both model and simulation results. This is not a surprising outcome and is just consistent with our intuition: if we refresh DHT routing tables more frequently, we may have more fresh and correct information, which ensures a higher lookup success ratio.

Influence of lookup rate

To analyze the influence of lookup rate, we change it from 1 times per peer's lifetime to 10 times per peer's lifetime while keeping other parameters as constants. Both model and simulation results are shown in Fig. 3.7.

It is surprising to see that, although we have already set the lookup rate large enough to 10 times per peer's lifetime, lookup success ratio still stays quite stable. This result again shows the scalability of DHT, but in a much larger degree comparing to that shown by system population. With an increasing system population, we claimed that scalability is gained from the in-

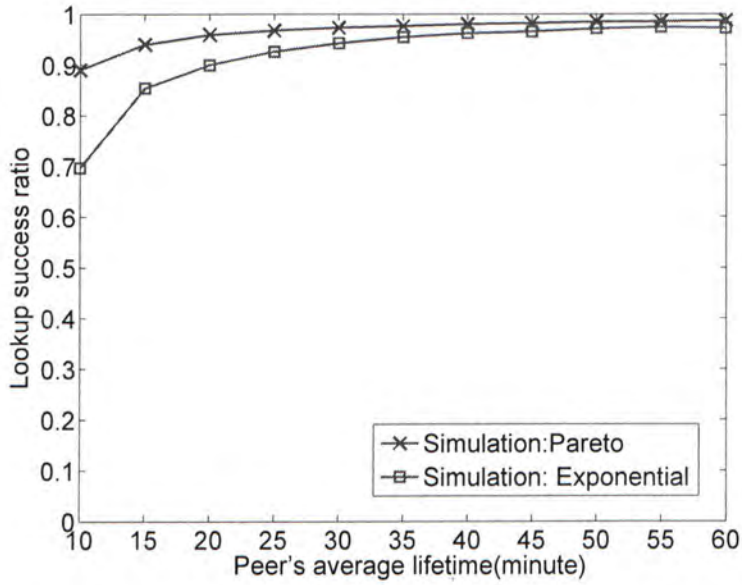
creasing number of peers sharing the burden. For the increasing of lookup rate, system workload increases while population is unchanged. Scalability in this part is gained from the fact that totally distributed DHT system almost holds no bottlenecks for workload [25].

3.4 Observations from modeling work

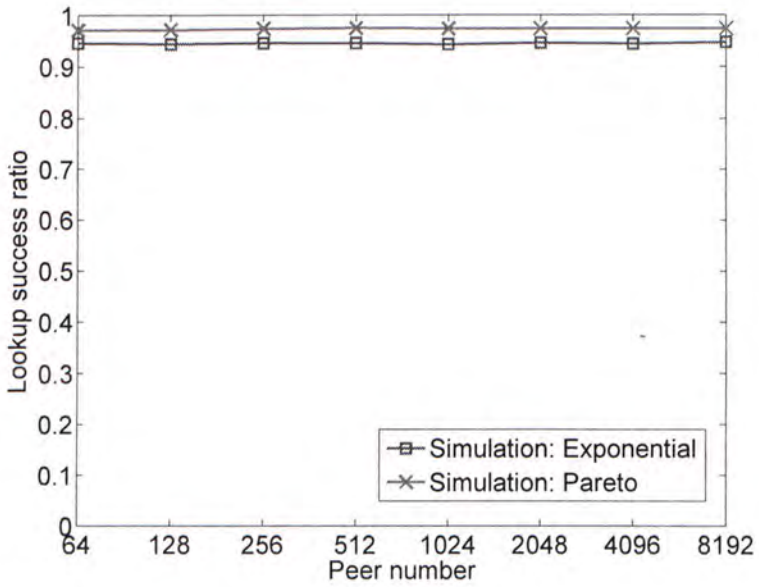
In last subsection, we set up simulation experiments to: 1) analyze parameters' influences to system reliability; 2) evaluate theory model's performance. From this work, we find that peer's lifetime and DHT stabilization interval influence system reliability more than system population and lookup rate do. Meanwhile, the fact that theory model results resembling simulation results very well shows that, although we haven't modeled all the details of DHT systems, we have captured the most important aspects in our model. It can also be seen as a solid basis for using simple equations to predict reliability of a DHT-based system. With this theory model, we will also be able to know how to choose system parameters (i.e. stabilization interval) to fulfill reliability requirements when we design a DHT-based system.

Besides of above applications, previous analysis also helps us to find out an interesting phenomena: the reliability (lookup success ratio) of system with exponential distributed peer lifetime is always an lower bound for that with pareto distributed lifetime (with the same mean value, as shown in Fig. 3.8). We analyze this phenomena theoretically on the basis of following corollary:

Corollary 3.1 *Under the same DHT system structure and with the same mean value* ($\frac{\beta}{\alpha-1} = \frac{1}{\lambda}, \alpha > 1, \beta > 0, \lambda > 0$)*, pareto*($\alpha, \beta$) *distributed lifetime can gain a higher single peer reliability and further a higher system reliability than exponential*(λ) *distributed*



(a) Different peer average life time



(b) Different peer number

Figure 3.8: Comparison of system performance of Exponential and Pareto distributed lifetime

lifetime.

Proof: According to Proposition 2.3, single peer reliability $p = \int_0^\infty Pr\{R \geq x\} \times f_S(x)dx$. With the same distribution of S and $x \geq 0, 0 \leq f_S(x) \leq 1, 0 \leq Pr\{R \geq x\} \leq 1$, p monotonously increases with $Pr\{R \geq x\}$. Meanwhile,

$$\frac{Pr\{R_{Pareto} \geq x\}}{Pr\{R_{exp} \geq x\}} = \frac{((\beta + x)/\beta)^{(1-\alpha)}}{e^{((1-\alpha)/\beta)x}} = \left[\frac{e^{\frac{x}{\beta}}}{(1 + \frac{x}{\beta})}\right]^{(\alpha-1)}$$
$$= \left[\frac{\sum_{n=0}^{\infty} \frac{1}{n!}(\frac{x}{\beta})^n}{(1 + \frac{x}{\beta})}\right]^{(\alpha-1)} > 1$$

always holds when $x > 0, \alpha > 1, \beta > 0$. Consequently, $Pr\{R_{Pareto} \geq x\} > Pr\{R_{exp} \geq x\}$ and further $p_{Pareto} > p_{exp}$ holds. According to Corollary 2.5, system reliability only depends on single peer reliability p under the same system structure, therefore above corollary holds. ∎

Intuitively we can understand above corollary like this: for DHT-based systems, peer's liveness is refreshed by stabilization. Instead of pure lifetime, the system can only observe peer's residual lifetime, which determines how long this peer will be alive from now on. For exponential distribution, it has the memoryless property and its residual lifetime is still exponential distributed with the same mean value. On the other hand, pareto distribution has long-tailed behavior, and its residual lifetime is still a pareto distribution, but with a larger mean value. This explained why performance of system with exponential distributed lifetime is always an lower bound for that with pareto distributed lifetime.

3.5 Methods of DHT stabilization

In previous discussion, we have assumed a constant stabilization interval in DHT algorithms. But in original designs of many

DHT algorithms, the way to do stabilization is not specified, or exactly defined. Generally speaking, one can specify stabilization interval to be a constant or a random value. For example, Chord's original version uses uniform distributed interval[4] and many modeling studies assume exponential stabilization intervals to simplify analysis. With all of this variations, few papers have discussed which one is optimal for DHT under certain conditions. The most related work in this subject is done by Z. Yao et al[12], but they focus on the isolation problem in Chord. In this section, we try to analyze the performance of different stabilization methods based on lookup success ratio, which is a more general metric for DHT system performance.

Similar to the analysis for Proposition 2.3, we provide following proposition for randomized stabilization.

Proposition 3.2 *Let $F_R(x)$ be the CDF of peer's residual lifetime R, $F_T(x)$ be the CDF of random stabilization interval T and $F_S(x)$ be the CDF of exponentially distributed lookup arriving interval $S(F_S(x) = 1 - e^{-\lambda' x})$, then single peer reliability is given by:*

$$\begin{aligned} p &= Pr\{R \geq S'\} = \int_0^\infty Pr\{R \geq x\} \times f_{S'}(x)dx \\ &= \int_0^\infty (1 - F_R(x)) \times \frac{f_S(x)[1 - F_T(x)]}{\int_0^\infty F_S(y)f_T(y)dy}dx \end{aligned} \tag{3.10}$$

where S' is the interval between lookup's arrival and last stabilization, and for Exponential(λ) distributed lifetime, we have:

$$\begin{aligned} p &= Pr\{R \geq S'\} = \int_0^\infty Pr\{R \geq x\} \times f_{S'}(x)dx \\ &= \frac{\int_0^\infty \lambda\lambda' e^{-(\lambda+\lambda')x}[1 - F_T(x)]dx}{\int_0^\infty (1 - e^{-\lambda' y})f_T(y)dy} \end{aligned} \tag{3.11}$$

Proof:

To get p, we need to decide the distribution of S'. Process is shown as follows:

$$Pr\{S' \leq x\} = Pr\{S \leq x | S \leq T\} = \frac{Pr\{S \leq x, S \leq T\}}{Pr\{S \leq T\}}$$

$$Pr\{S \leq T\} = \int_0^\infty Pr\{S \leq t\} f_T(t) dt = \int_0^\infty F_S(t) f_T(t) dt$$

$$\begin{aligned} Pr\{S \leq x, S \leq T\} &= \int_0^\infty Pr\{S \leq x, S \leq t\} f_T(t) dt \\ &= \int_0^x F_S(t) f_T(t) dt + \int_x^\infty F_S(x) f_T(t) dt \\ &= F_S(x)[1 - F_T(x)] + \int_0^x F_S(t) f_T(t) dt \end{aligned}$$

Hence we have

$$Pr\{S' \leq x\} = \frac{F_S(x)[1 - F_T(x)] + \int_0^x F_S(t) f_T(t) dt}{\int_0^\infty F_S(t) f_T(t) dt}$$

By differentiating both sides we have

$$f_{S'}(x) = \frac{f_S(x)[1 - F_T(x)]}{\int_0^\infty F_S(t) f_T(t) dt}$$

together with Lemma 2.2, above proposition holds.

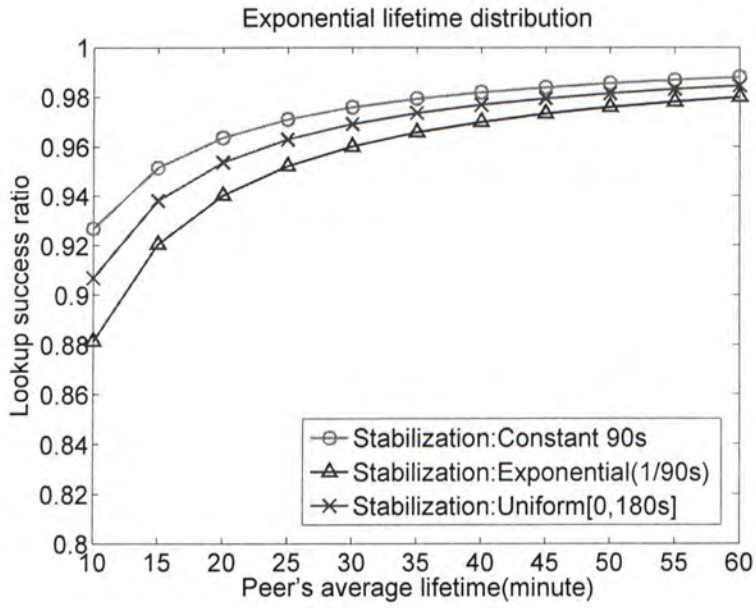
■

Corollary 3.3 *Assume stabilization interval is exponential*$(1/T)$ *distributed, then single peer reliability is given by:*

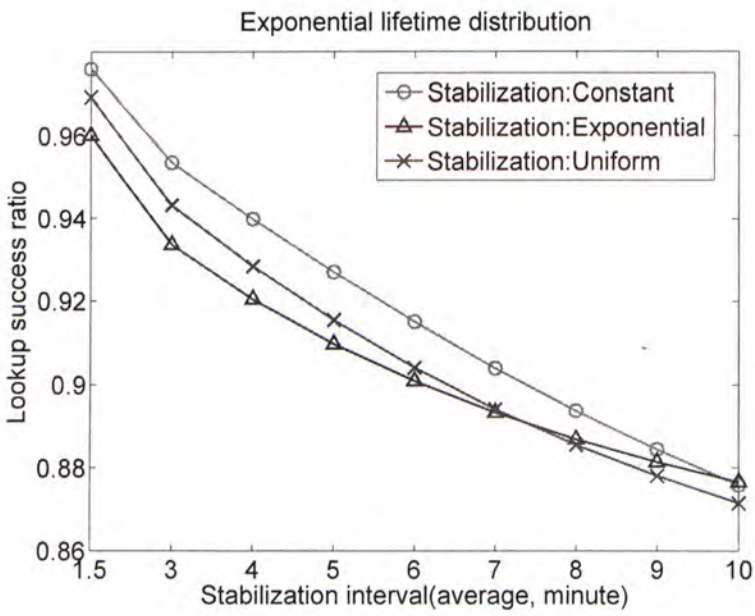
$$p = \frac{\lambda' T + 1}{(\lambda + \lambda')T + 1} \tag{3.12}$$

and for uniform$[0, 2T]$ *distributed stabilization, single peer reliability is given by:*

$$p = \frac{(\lambda')^2[2(\lambda + \lambda')T + e^{-2(\lambda+\lambda')T} - 1]}{(\lambda + \lambda')^2[2\lambda' T + e^{-2\lambda' T} - 1]} \tag{3.13}$$



(a) Influence of peer's lifetime



(b) Influence of stabilization interval

Figure 3.9: Numerical results: different stabilization methods

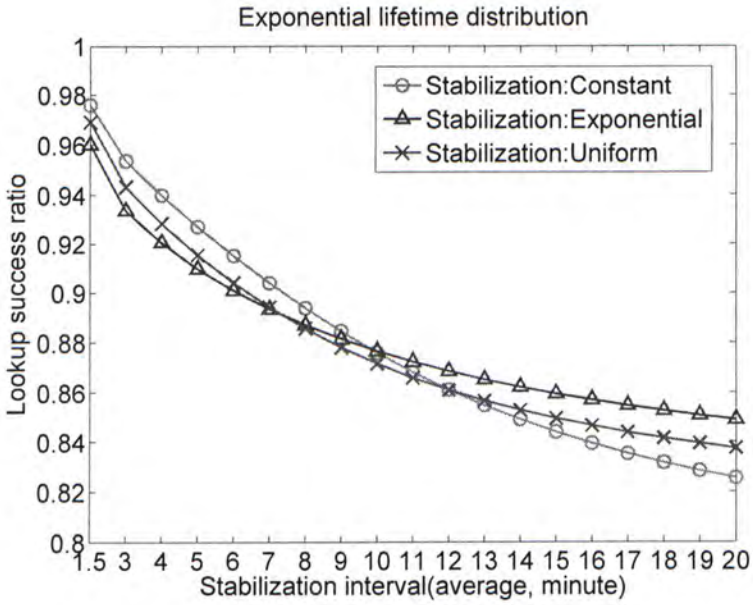
Numerical results of Corollary 3.3 are shown in Fig. 3.9 where parameter are selected to be in the same range as in previous theory model and simulation experiments. From these numerical results, two main observations can be derived:

- $E[T]$ influences system performance more than T's distribution $F_T(x)$ does (see Fig. 3.9(b)).
- Within our parameter range, constant stabilization always achieves the best performance and exponential distributed stabilization's performance is the worst.

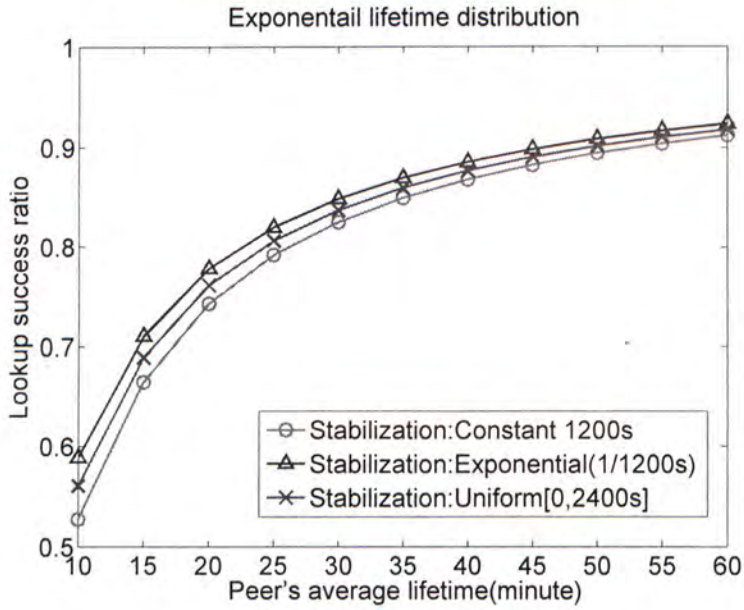
Although we have chosen parameter ranges based on the considerations of real world P2P applications, above numerical results still trigger us to think more about the relationship (related to system performance) between $E[T]$ and $F_T(x)$, based on following questions:

- Which one decides performance of stabilization? The stabilization interval distribution itself or the mean value?
- Does constant stabilization always be the optimal?
- Once the mean value of stabilization interval is specified, can we determine the optimal method for stabilization?

We answer above questions by comparing experiment results shown in Fig. 3.10. We may see from Fig. 3.10(a) that, keeping other parameters unchanged, for small average stabilization interval, constant stabilization is the optimal; and for large average stabilization interval, exponential distributed stabilization is the optimal. Hence we may say that the optimal method for stabilization changes with its mean values. For the methods utilized a lot in real P2P application, i.e. small stabilization interval, the optimal method is constant stabilization. Zhongmei Yao et al [39] derive a similar result which says that, as mean



(a) Influence of stabilization interval: large range



(b) Influence of peer's lifetime: large stabilization interval

Figure 3.10: Numerical results: different stabilization methods(large interval)

stabilization interval $E[T] \to 0$, node isolation probability under constant stabilization interval is no greater than that under any random stabilization interval. We believe their result is partially consistent with our results.

For the other two questions, by comparing Fig. 3.9(a) and Fig. 3.10(b) we may see that, in Fig. 3.9(a) it is always the constant stabilization to be the optimal and in Fig. 3.10(b) it is always the exponential distributed stabilization to be the optimal. Hence, we may say that, keeping other parameters and network condition unchanged, once we deicide the average stabilization interval, optimal method is also determined (or at least within our range of parameters, it is determined).

□ **End of chapter.**

Chapter 4

A Black-Box Study of Xunlei

1 An Overview of Xunlei and its key components

1.1 An overview

As claimed by the company and also as observed from our experiments (details will be discussed later), Xunlei supports different methods for file downloading, including BitTorrent, eDonkey, Kad, FTP and HTTP. These methods combined together, become Xunlei's multi-protocol strategy for file downloading. Fig. 1.1 shows a global view of Xunlei, which briefly introduces its system structure.

Nowadays there are mainly two popular P2P file downloading networks which represent a majority of P2P file downloading application users, namely BitTorrent network and eMule network. While these two networks share many similarities in their P2P technologies, each has its own special designed properties. P2P users are more likely to use BitTorrent for sharing and downloading popular files, and to use eMule for searching and downloading rare resources. Xunlei has embedded both of them into its own client, which saves its users for having to run two different clients for different purposes. As shown in Fig. 1.1, a Xunlei client can participate in both these two networks by

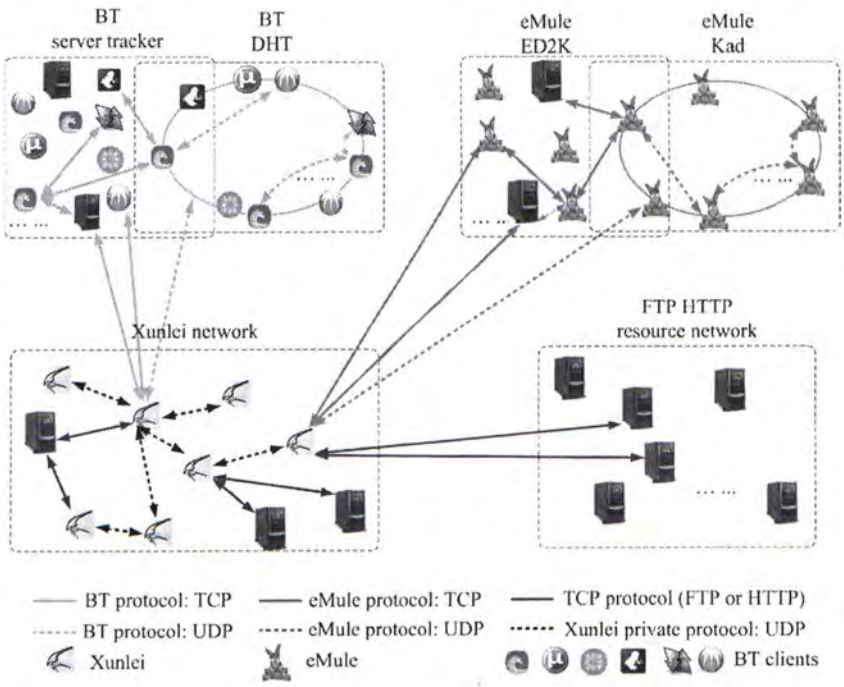


Figure 1.1: Xunlei overview

communicating with BT clients or eMule clients using their own protocols respectively.

Besides these P2P networks, another popular, and also traditional way for users to get resources is via FTP or HTTP. The maturity of multi-threads and resuming downloading technologies makes it possible for a client to download a file in client-server and P2P ways at the same time, which provides a solid foundation for Xunlei's multi-protocol downloading method. We believe Xunlei uses a certain mechanism to combine file parts (downloaded from different sources) into a complete file. But at present we are not clear about the details.

Irrespective whether it is BitTorrent network, eMule network, FTP or HTTP network, all of them already exist and are well developed for a long time. A Xunlei client participating in these network is more like a guest or a foreigner: it needs to speak different languages if it wants to stay in different countries (net-

works). Then we may ask: what is Xunlei's mother tongue?

We find that Xunlei clients also form a Xunlei network, where they communicate with each other using Xunlei private P2P protocol (as their mother tongue). This protocol is based on UDP, with special designed error control and congestion control mechanisms, and it is frequently used for exchanging peer lists and file data during file downloading process. We will discuss the details in section 4.

1.2 Key components

Connections: During the run of Xunlei, a client opens several TCP and UDP connections. TCP connections are used for: 1) contacting Xunlei servers to get resources used in user interface and to post user information; 2) contacting BitTorrent's tracker server to get file information (if needed); 3) exchanging file data with a BitTorrent/eMule client based on BitTorrent/eMule protocol, or downloading files from FTP servers. UDP connections are used for: 1) exchanging peer lists and file data (based on Xunlei's private protocol); 2) searching file information or swarm peers in DHT network (can be BitTorrent or Kad network).

Ports: Usually three UDP ports are pre-assigned upon installation of a Xunlei client. One is for exchanging peer lists and file data (based on Xunlei's private protocol); the other two are for listening to Kad and eDonkey networks[1]. In addition, a new UDP port is randomly assigned for BitTorrent DHT lookup packets (if needed). For TCP connections (used to contact servers or to exchange messages with BitTorrent/eMule clients[2]), ports are usually randomly assigned.

User interface components: Xunlei's user interface contains several parts: advertisements, movie recommendations, hot movie list etc. A number of servers are deployed to de-

[1]Kad and eDonkey are networks supported by eMule clients.

[2]Both BitTorrent and eMule protocols use TCP for data transmission.

liver these information. According to our observations, Xunlei servers have different responsibilities. There are usually several servers specially in charge of a particular function and usually these servers are located geographically near each other.

Downloading history cache: Each Xunlei client keeps a record of its own downloading history. Besides the downloaded list shown in the user interface, clients also store BitTorrent's meta files (.torrent, .bt.dat and .bt.cfg) in their local cache if they have performed any BitTorrent downloading tasks. Later, this downloaded list and cached BitTorrent files can be used for resource sharing.

2 Participating into other swarms: Xunlei's multi-protocol downloading strategy

2.1 BitTorrent and eMule basics

BitTorrent was designed by Bram Cohen in April 2001 and the first implementation was released on July 2, 2001 [4]. Subsequently numerous (more than 50) clients have been developed by different organizations based on the original BitTorrent protocol [3] and they form the **BitTorrent network**. While these clients each has its own features, they can communicate with each other using the original BitTorrent protocol as a common language. In BitTorrent network, file downloading processes are standard, by following rules and sequences: 1) downloading the metainfo file (.torrent) from torrent-discovery sites[3] and getting tracker information form it; 2)finding swarm peers from server tracker (via TCP) or serverless DHT tracker (via UDP); 3) exchanging file chunks with other swarm peers based on predefined messages (via TCP): *choke, unchoke, interested, not interested, have bitfield, request, piece, cancel* [21].

[3]Mininova, Private Bay .etc

Unlike BitTorrent, eMule supports two networks: **eDonkey2000 (ED2K) network** and **Kad network**. It represents a large number of users in both ED2K network (more than 90 percents) and Kad network (more than 95 percents, together with aMule and MLDonkey). Hence eMule can be seen as a representative for both of them.

In ED2K network, clients connect to servers to search files and swarm peer information. Instead, clients in Kad network use DHT (Kademlia [6]) to fulfill these functions. It is interesting to note that most of Kad users are also connected to ED2K servers, which means Kad and ED2K networks overlap a lot. In other words, they are serving the same target users by providing similar services in different manners: ED2K is based on servers [17, 5] and Kad is based on DHT (and hence is totally distributed). Once swarm peers are found and corresponding client-to-client connections are established, eMule does not differentiate ED2K and Kad users: file exchanging always follows the same rule.

2.2 BitTorrent and eMule in Xunlei

To study BitTorrent and eMule behaviors in Xunlei, we use a Xunlei client to perform several BitTorrent and eMule downloading tasks respectively. For each task, we first delete all the files about Xunlei, reboot the PC and install Xunlei again. Then we start the task and use Wireshark to monitor the whole downloading process. We close Xunlei immediately after the file is completely downloaded.

By claiming a task to be a BT task, we first download a .torrent file and then use Xunlei to open it and to download the file contained in it. By claiming a task to be an eMule task, we observe an ED2K link (same as those used in eMule clients) in Xunlei's downloading task manager.

Table 4.1: Examples of BT tracker servers that Xunlei clients contact

IP address	Host	Response
121.14.243.98	tk.greenland.net	none
121.14.243.99	tk2.greenland.net	none
218.16.124.111	bt.romman.net	return request interval and binary data(should be peer list)
222.73.173.113	btfans.3322.org	redirect me to 218.202.227.27:8085
220.189.250.104	bt.ktkj.com bt.ai-sky.com	failure(unregistered)
221.130.196.76	share.comoe.cn	failure(unregistered)
218.202.227.27	unknown	exchange TCP messages (should be peer list)

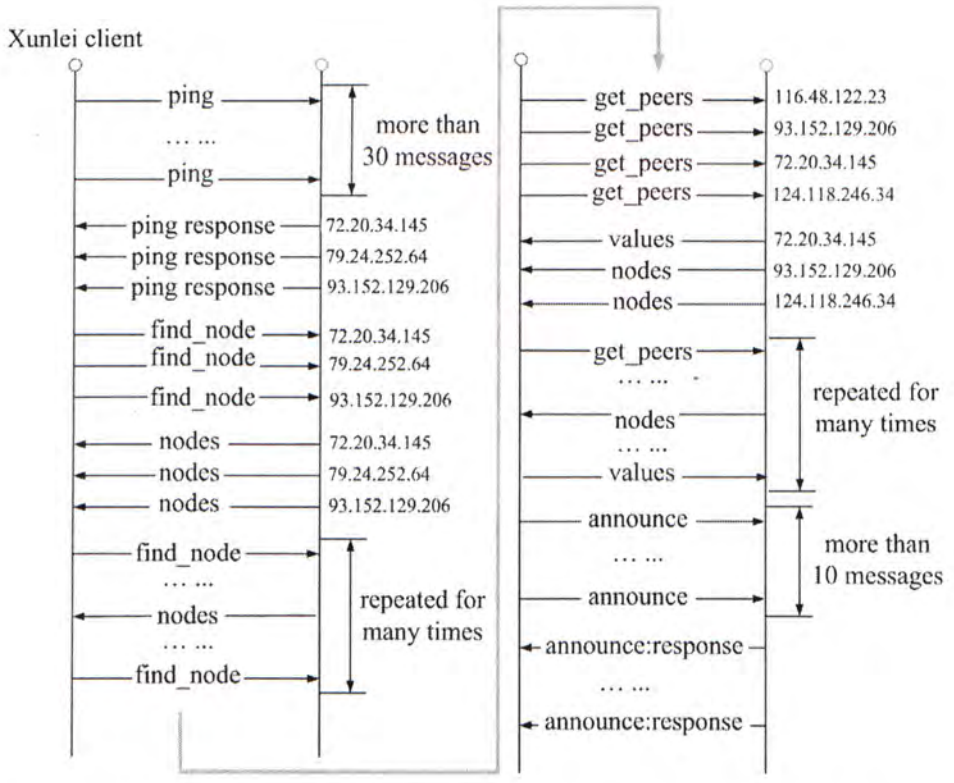


Figure 2.2: A BitTorrent DHT flow captured in a Xunlei client

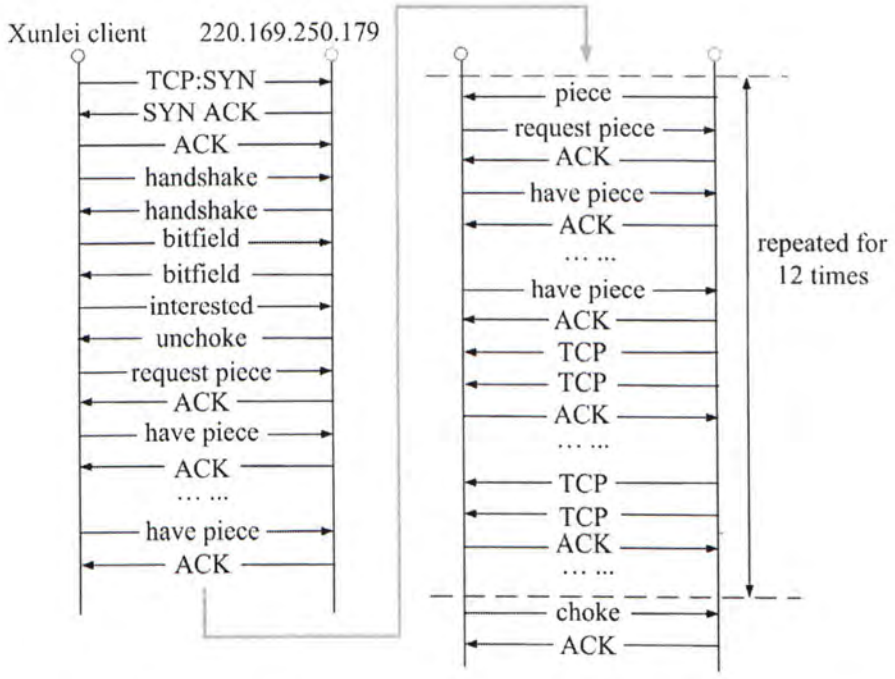


Figure 2.3: A BitTorrent client-to-client file exchange flow captured in a Xunlei client

We observe that our Xunlei client fully performs BitTorrent functions (as discussed in last subsection) during the file downloading process. We find a number of BitTorrent tracker servers that our Xunlei client has contacted[4] (as shown in Table.4.1), and some UDP flows with message types consistent with BitTorrent's DHT design (as shown in Fig. 2.2). We also find that our Xunlei client has exchanged file data and control info with other clients based on BitTorrent protocol (an example is shown in Fig. 2.3). We may conclude that, in this part Xunlei follows the original design of BitTorrent quite well.

During the downloading process of an eMule task, our Xunlei client performs a number of eMule steps, including using Kad

[4]It is interesting to note that only some of these tracker servers have replied our Xunlei client's requests: two of them replied nothing and another two of them rejected our client directly. We believe the "no response" is due to the heavy work load of those tracker servers, and the rejection comes from the fact that our client is unregistered (as implied by the http messages we received.)

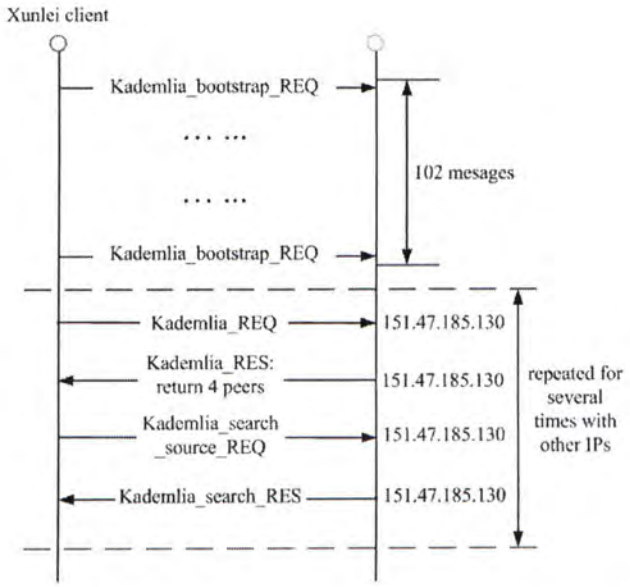


Figure 2.4: A Kad flow captured in a Xunlei client

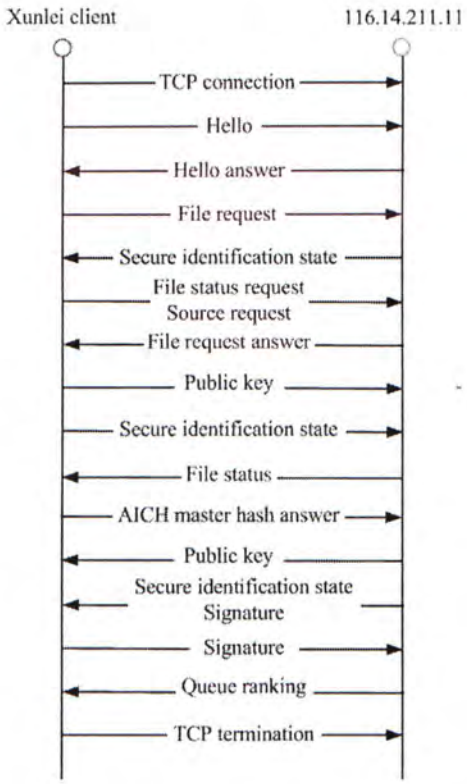


Figure 2.5: An eMule client-to-client file exchange flow captured in a Xunlei client

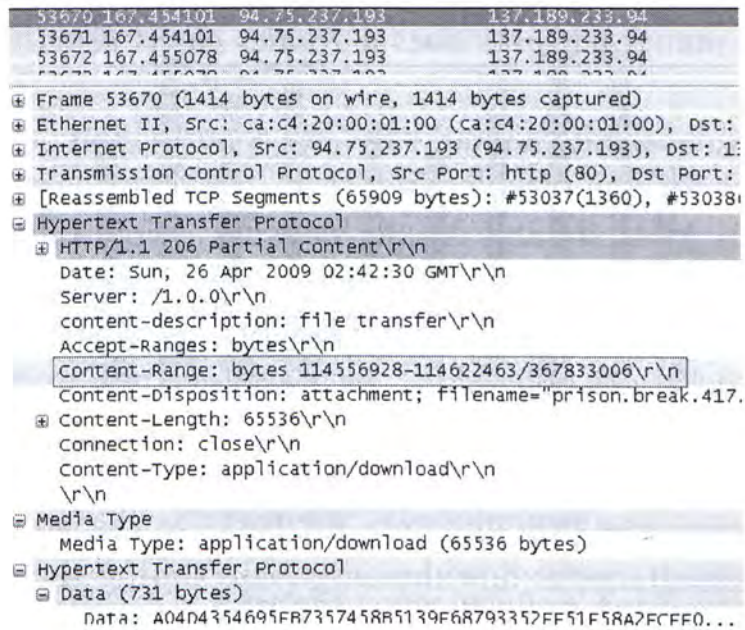


Figure 2.6: Getting a file part via HTTP

to search swarm peers and using eMule protocol to exchange file data and control info. We show them in Fig. 2.4 and Fig. 2.5 respectively. In this part, Xunlei follows eMule's original design quite well. More details about eMule can be found in [17].

2.3 Multi-protocol downloading

From the subsections above, We know that Xunlei can communicate with different clients from different networks using different protocols, and we have observed real flows related to BitTorrent and eMule protocols in Xunlei's downloading process. Since the mechanisms used in FTP and HTTP are well understood, we omit real flow demo about FTP or HTTP here. Generally speaking, by claiming a particular binary range, a Xunlei client can download any part of a file as it wants via FTP or HTTP. We show a packet snapshot of this process in Fig. 2.6. To better understand the multi-protocol downloading, we further discuss the following questions:

How many networks does a Xunlei client usually participate in?

A Xunlei client can and will always participate in at least two networks for one downloading task: the one where the original source is located[5] and Xunlei network. For some tasks (not all of them), we may also find a Xunlei client participates in more than two networks. For example, for some BitTorrent or eMule tasks, a Xunlei client may also get resources from FTP or HTTP servers at the same time. We believe Xunlei should adopt some multi-threads technologies, which helps its client collect resources from different networks at the same time.

For simultaneous downloading tasks during the same run of a Xunlei client, the extreme case is that it may participates in BitTorrent, eMule, FTP and HTTP networks at the same time when these tasks are of multiple types (BT, eMule and non-P2P) during one run.

Does a Xunlei client fully perform all the functions when it participates in another network besides its own network?

We believe it does. As shown in last subsection, when participating in BitTorrent network, a Xunlei client acts exactly like a BitTorrent client: it contacts tracker servers, it performs DHT lookups and it communicates with other BitTorrent clients in original BitTorrent protocol. And same situation happens when it participates in eMule network.

Does a Xunlei client have a preference for which protocol to use?

Although we have not done a measurement to answer this question, we observe from previous experiments that a Xunlei client gets a large amount (sometimes even a majority) of a downloaded file from its own network, especially when downloading tasks are targeting at Chinese users (as shown in Fig. 2.7)[6].

[5]For a BT (eMule) task, original source is located in BitTorrent (eMule) network and for a non-P2P task, it is located in FTP or HTTP network.

[6]The 10 torrents are the same as those used in Table. 1.1: the first five are downloaded

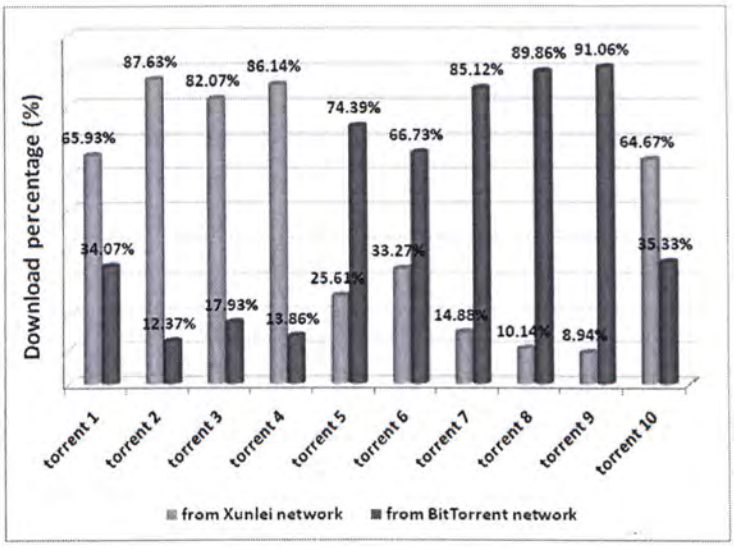


Figure 2.7: Examples of download percentage from Xunlei and Bittorrent networks for BT tasks

Instead of well-developed P2P networks, why does a Xunlei client get a large amount of resource from its own network? Recall Table. 1.1 we may further see that this choice reaches high downloading speed for Xunlei users. We conjecture that either Xunlei uses certain methods (or deploys many servers) to help its clients find more swarm peers (means more sources to get the target file), or it designs some algorithms to improve the performance of file exchanging process. Both of these two parts are related to Xunlei's private protocol and we will discuss them in the following sections.

3 Xunlei servers

Xunlei deploys a number of servers to perform certain functions, such as delivering recommendation and advertisements used in user interface, broadcasting desktop news, checking software updates, collecting peer information and so on. Instead of finding

from Gougou search (Xunlei's entertainment resources search engine), where Chinese users are the targets.

all the servers deployed by Xunlei (which is unnecessary and also a heavy work), we are more interested in classifying their functions, both in startup and file downloading processes.

We use wireshark to monitor the whole process of opening a Xunlei client, downloading a file and closing the client. To get a general understanding, we repeated this process for many times.

We summarize the information of some servers we captured in Table. 4.2, including their IP addresses, locations and functions. Most of these servers are located in different areas of China. For a certain function, its corresponding servers are usually located geographically near each other, e.g. servers that are responsible for dealing with clients' POST messages are usually located in Guangdong.

Table 4.2: Information about some Xunlei servers

IP addresses	Location	Function(s)
58.251.57.86 ; 58.251.57.73	Shenzhen	delivering desktop news
221.203.179.6; 221.203.179.7	Liaoning	
60.19.64.46 ;60.19.64.52	Liaoning	delivering recommendation
60.19.64.62 ; 60.19.64.36		and news
60.28.15.204 ; 60.28.15.208	Tianjin	checking updates and
60.28.178.196 ; 60.28.178.205		virus scan
218.57.144.53 ; 218.59.144.53	Shandong	delivering advertisements
58.254.134.204 ; 58.254.134.205	Guangdong	dealing with clients' POST
58.254.134.206 ; 58.254.134.208		messages
58.254.134.209 ; 221.4.246.73		
58.251.57.201	Shenzhen	collecting and delivering
60.28.13.153 ; 60.28.178.207	Tianjin	requested file info
60.28.178.224; 125.39.72.105		

Based on our observation, we classify Xunlei servers into three categories: *initialization servers*, *POST servers* and *target file responsible servers.*

Initialization servers are contacted by Xunlei clients during

startup process to get resources used in user interfaces. Some of these servers are also hosts of Xunlei Portal, which shows the variety of one single Xunlei server's functions. Besides initialization servers, a Xunlei client also contacts several POST servers during startup process, based on HTTP POST messages[7]. Contents of these messages are of binary type (hence can not be decoded straightforwardly) and they are usually of different sizes. We conjecture that these POST servers may be used for collecting client information, as login servers. During the downloading process, a Xunlei client further contacts several other servers for delivering and retrieving file information. For different downloading tasks, a client usually contacts different servers. That is why we call them *target file* responsible servers. To support a Xunlei client, these three types of servers need to cooperate with each other to fulfill all the needed functions.

4 Understanding Xunlei's private protocol

In this section, we give our results of the analysis of Xunlei's private protocol, including but not restricting to its message types, data structures, error control and congestion control mechanisms. Due to the fact that Xunlei is a proprietary protocol, our results are based on the analysis of binary data. Hence all the results given here are conjectures, rather than official descriptions.

4.1 Exchanging peer lists

Besides existing methods (via tracker server or DHT lookup), Xunlei also uses its own UDP-based messages for exchanging peer list. We first show its message structure in Fig. 4.8.

[7]That's why we call these servers "POST servers".

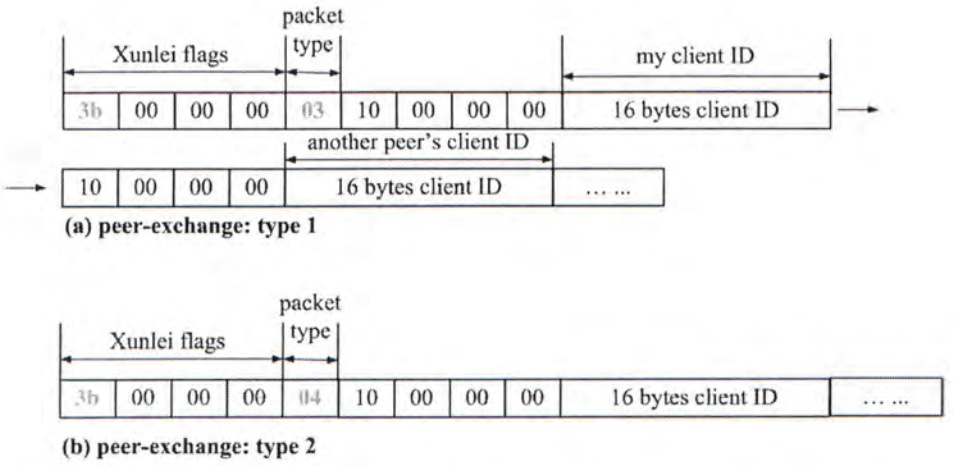


Figure 4.8: Xunlei's message structures for exchanging peer lists.

The first four bytes should contain the flags for Xunlei messages and the fifth byte specifies the message type. The following 4 bytes (10 00 00 00) probably serve as a 'separator' for 16 bytes client IDs. We conjecture that this type of message is used for exchanging swarm peers, for the reason that it contains other clients' IDs, and upon receiving it, our client immediately sends a handshake message to another peer whose client ID is exactly the same as the one attached in it. We believe that the 16 bytes client ID should contain information of corresponding peer's IP address and port number, which may be gotten by hashing them based on a particular hash function. Hence with this ID, our client can directly contact that peer.

We also find that there are two types of messages for peer exchange, with type number equals to 03 and 04 respectively. At present we are not quite clear about their differences, but this will not affect the following analysis. We guess type 04 message may be a simple version of type 03 message, since it ignores packet sending peer's client ID in the data structure.

Usually a Xunlei client keeps on exchanging this message with many IP addresses throughout the whole downloading process. We are not sure whether these IP addresses represent normal

peers or deployed servers. But we find that, most of the UDP ports at the other side are 8000. Unlike what we expected, we did not find any periodic behavior in this peer exchange process.

4.2 Exchanging file data

After finding a swarm peer, a Xunlei client directly establishes a UDP connection with it to exchange file data. We discover several specific exchanges during this process and we show them in Fig. 4.9. It should be noted that we don't capture all of these messages in a session, rather it is a summary from multiple sessions throughout all our experiments.

Handshake and handshake ACK: handshake (31 bytes) is usually performed at the beginning of a client-to-client connection. Upon receiving it, the other client replies a 39 bytes handshake ACK. A handshake message contains some basic information about the client and the connection, i.e. session flag, peer ID and packet ID[8].

Session flag is used for specifying a particular session (client-to-client connection) and its direction. In Fig. 4.9, 00 00 dc 97 and dc 97 00 00 are used respectively for the two directions of one connection. The 4 bytes peer ID is used to specify a particular peer, or client. It is unique for the same peer, even in different sessions. Packet ID is used to specify a particular packet, which will be frequently used later for exchanging file data.

Self-introduction: this message (29 bytes) is usually performed before or right after handshake message, for introducing 16 bytes client ID to specify a client. After this self-introduction message, all subsequent packets will only use 4 bytes peer ID to do this specification. We conjecture that Xunlei utilizes this method to reduce the length of its packet header. Session flag

[8]All these field are named based on our conjectures.

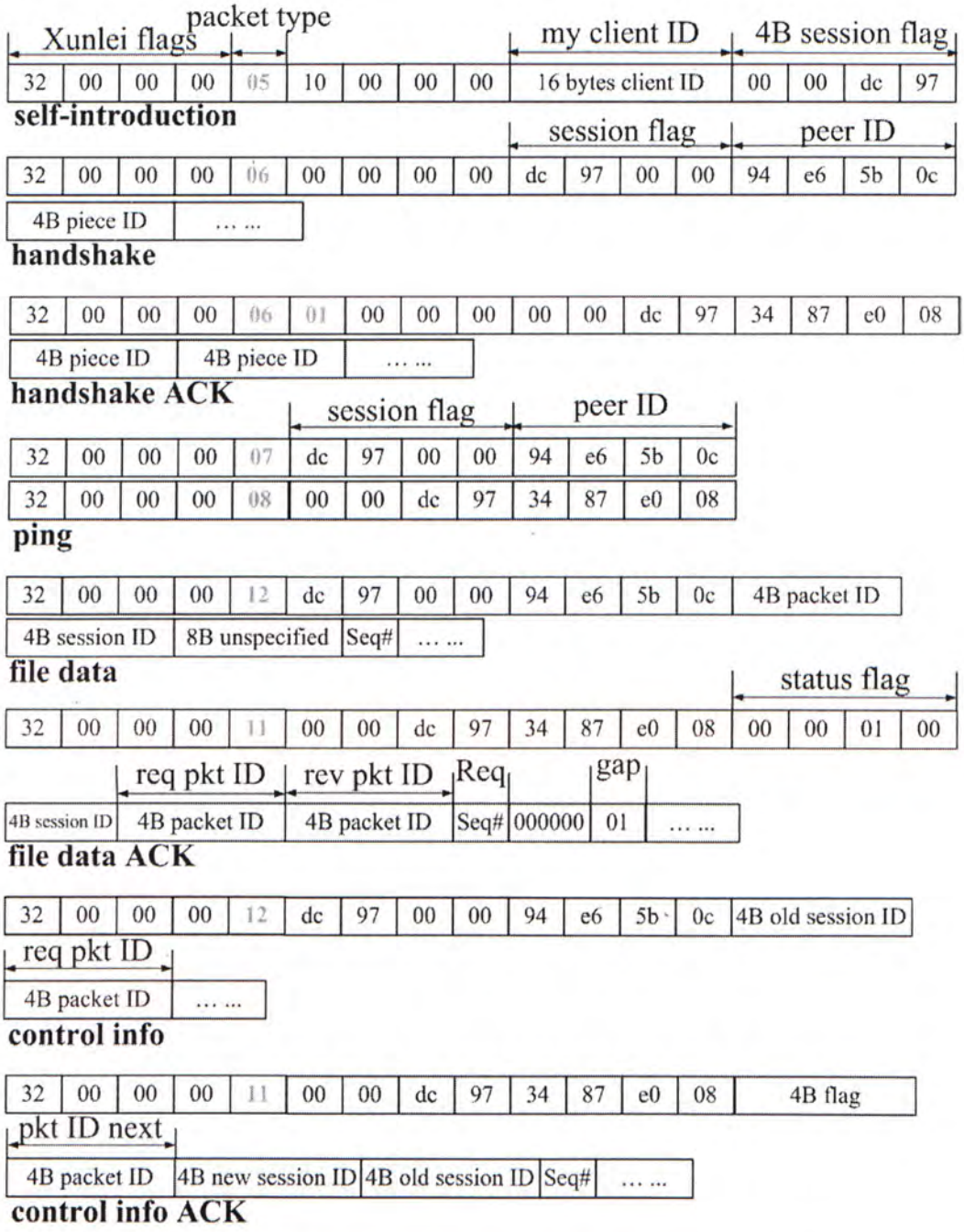


Figure 4.9: Xunlei's message structures for exchanging file.

is the same as that used in handshake messages.

Ping: this message (13 bytes) is performed when one client wants to re-connect to another client. It is quite light-weighted since it only contains the information of session flags and peer ID.

File data and its ACK: typically a file data packet is over 1000 bytes, with a few exceptions which should correspond to pieces at the end of a file. Besides session flag and peer ID, this packet also contains a field we refer to as 'session ID', which seems unique for a particular session, but it can be changed by control info (will be discussed later). We conjecture that it may be used for specifying a particular session (client-to-client connection), or for doing some security-related operations, such as error detection. The field 'seq#' (sequence number) is used to organize the sequence of packets.

Upon receiving a file data packet, the receiver replies an ACK (usually of 37, 38 or 39 bytes), with the information of received packet ID, requesting packet ID, requesting packet seq# and gap size. This ACK message also contains a field of status flag, showing whether there is a gap in receiver's buffer or not. The gap is generated by out-of-ordered packet arriving (a common issue for UDP transmission) and gap size is calculated as *largest packet_rev_seq# − packet_req_seq#* (the largest sequence number of the received packets minus that of the requested packet).

Control info and its ACK: Control info messages (89 bytes) share the first 13 bytes common header with file data messages, which is followed by a 4 bytes old session ID and another 4 bytes requesting packet ID. Upon receiving this control message, the client replies a 37 bytes ACK claiming the new session ID and the packet ID that will be sent next. Session ID will be changed to the new one in all subsequent packets, until the client receives another control info.

In Fig. 4.10, we show an example of Xunlei's private protocol

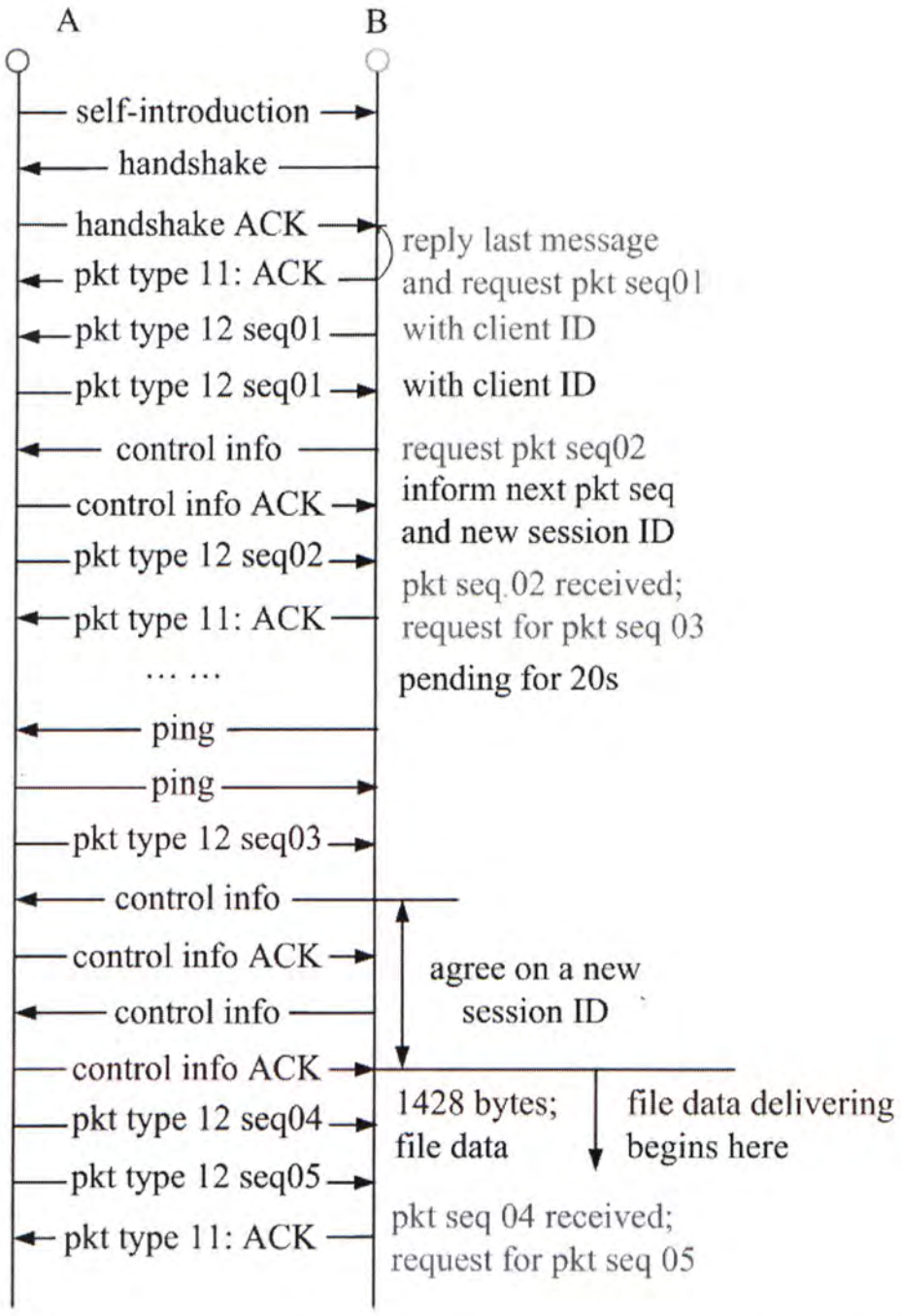


Figure 4.10: An example for file data exchange process of Xunlei's private protocol

flow, which introduces the typical process for Xunlei clients to exchange file data: 1) sending self-introduction and exchanging handshake messages to provide/retrieve information like session flag, client ID and peer ID; 2) using control info to make an agreement on session ID; 3)delivering file data or sending ACKs to request packets.

4.3 Error control and congestion control

A common issue for all UDP-based applications is that, while they take advantage of UDP's light-weight and flexibility, they need to design their own application-level error and congestion control mechanisms. Usually these mechanisms are performed via ACK messages. According to our observation, Xunlei adopts several mechanisms for uploaders to deal with ACKs and to decide packet sending policies. We consider them as Xunlei's error and congestion control mechanisms

The way a Xunlei client sends an ACK message is different from that used in TCP or Go-Back-N (GBN). Here ACKs do not need to be sent in order: a client will send an ACK immediately after receiving a packet, no matter whether this packet arrives in order or out of order. So in some sense Xunlei's ACK is more like that used in Selective Repeat (SR) [7], except that it provides more information, such as requesting packet seq# (negative acknowledgement), which can be used for retransmission.

Based on the above understanding, we make some conjectures about Xunlei's error control and congestion control mechanisms. We list them as follows and use a real flow captured in Xunlei's file data exchange process shown in Fig. 4.11 as an example to explain them.

- Xunlei will do retransmission, either upon receiving **triple duplicate ACKs** or after a **timeout**.

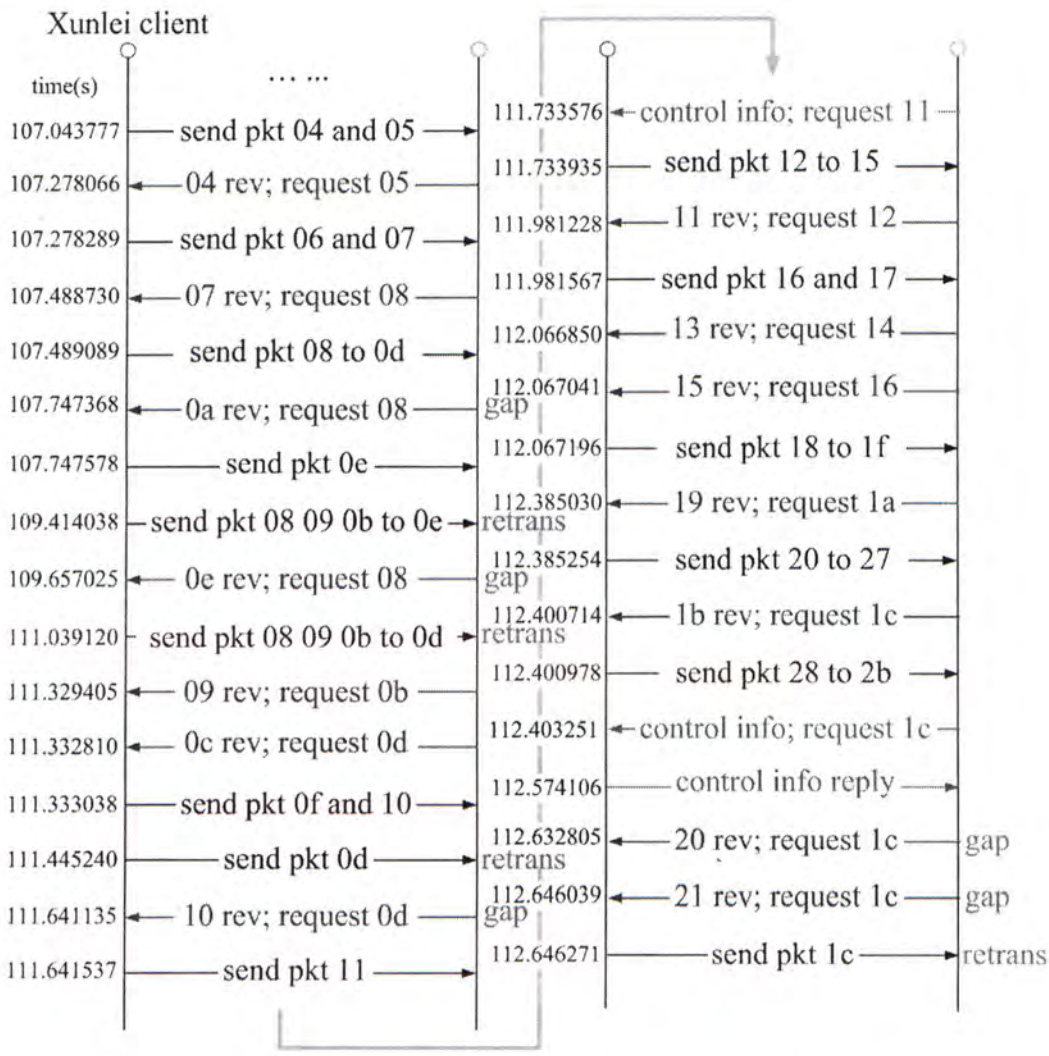


Figure 4.11: An example for Xunlei's error control mechanism

To recognize an ACK to be a duplicate ACK, the receiver needs to confirm that: 1)this ACK requests for the packet with the same seq# as last ACK has requested for; 2) this ACK shows a non-zero gap size. Hence in Fig. 4.11, our client retransmitted packet 1c immediately after receiving three duplicate ACKs[9].

Besides triple duplicate ACKs, retransmissions can also be triggered by timeout (similar to TCP). In Fig. 4.11, our client retransmits packet 08 at 109.414038s and packet 0d at 111.44524s respectively, both due to the timeout, rather than triple duplicate ACKs. According to our observation, this timeout is around 1.5 seconds.

- During a retransmission process, Xunlei will retransmit all the packets falling in the gap [**pkt_req, pkt_rev(largest)**], except those that have already been ACKed.

This retransmission mechanism is similar to that used in Selective Repeat (SR). In Fig. 4.11, our client retransmits packet 08 09 0b 0c 0d and 0e at 109.414038s, without packet 0a since 0a has already been ACKed at 107.747368s. This mechanism can reduce the work load and avoid unnecessary retransmissions.

Besides error control, another interesting question is whether Xunlei provides congestion control to avoid overflowing the network and flow control to avoid overflowing receiver's buffer. We show one Xunlei UDP upload flow in Fig. 4.12 to study and explain Xunlei's congestion control mechanism.

From Fig. 4.12, we find two typical behaviors of Xunlei's data sending policies, which share some similarities with those used in TCP and are also reasonable for all congestion control designs: 1)when there is no retransmission (detection of error), data sending rate will be increased (see 15s to 20s in Fig. 4.12); 2) when there are retransmissions (indicated by duplicated packet ID),

[9]During the calculation of duplicate ACKs, the one embedded in 89 bytes control message always accounts one.

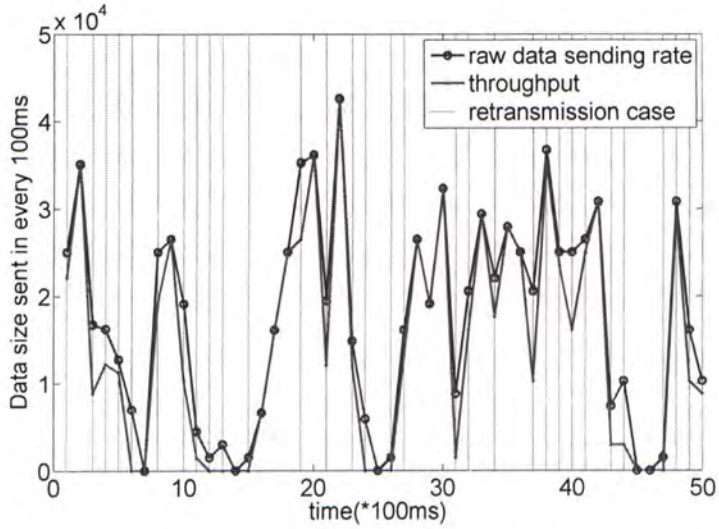


Figure 4.12: A Xunlei UDP upload flow

data sending rate will be decreased (see 4s to 8s in Fig. 4.12). It is not clear how Xunlei implements the increase and decrease functions, or whether a window size is used for flow and congestion control. Meanwhile, the error control mechanism used in Xunlei seems to trigger flow control actions: the retransmission triggered by triple duplicate ACKs or by timeout can be used to slow down the sending rate, and hence avoids overflowing receiver's buffer.

5 Further discussions

In this section, we try to speculate on the reasons for Xunlei's good performance.

5.1 Proximity of content

The first guess is the proximity of content. Xunlei is originated from China and designed for mainly Chinese users. If true, it means most users are "close" to the content they try to download. Does this mean Xunlei will lose its speed advantage when

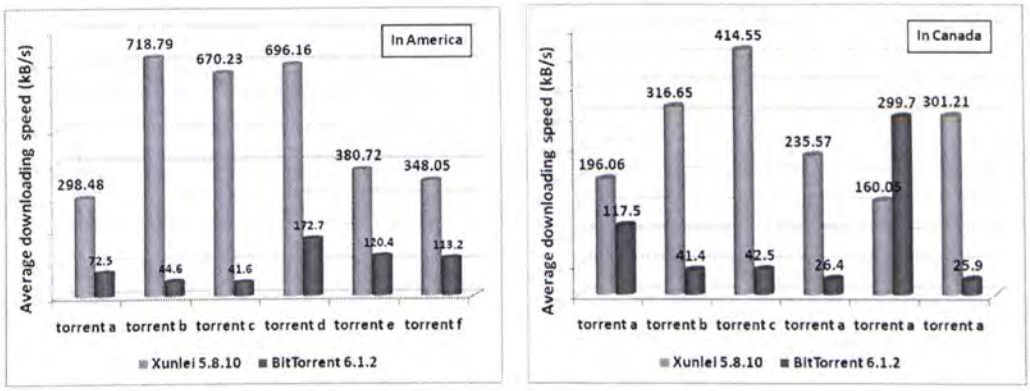


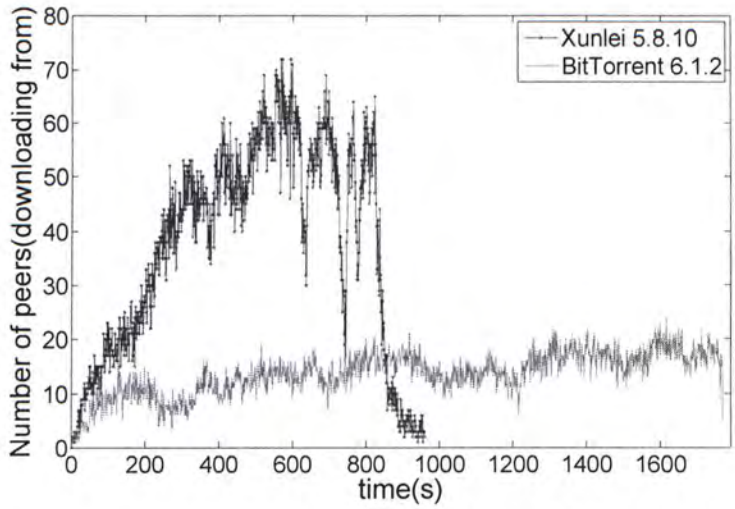
Figure 5.13: User experiences of file downloading speed (outside China)

it is used outside China?

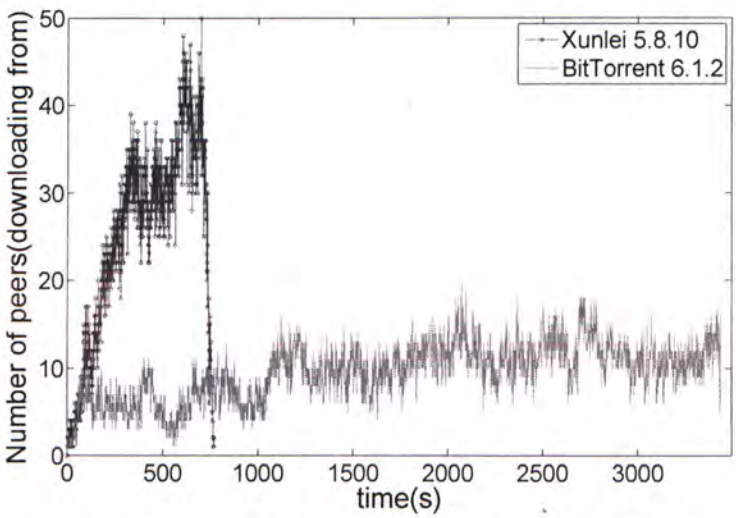
The answer is no. We performed several experiments in Canada and America respectively. These experiments are performed in April 09. Each time we use a Xunlei client (Xunlei 5.8.10) and a BitTorrent client (BitTorrent 6.1.2, installed in the same PC) to download the file contained in the same torrent one after another (to minimize the time influence). We repeated this process with six different torrents. The first three torrents (torrent a-c, as shown in Fig. 5.13) are discovered via gougou search [13] (targeting at Chinese users) and the other three are discovered via Mininova (a popular English-language torrent-discovery site). Results (average downloading speed) are shown in Fig. 4.13(a) and Fig. 4.13(b), which indicate that Xunlei still achieves high downloading speed when it is used outside China.

5.2 Active swarm peers

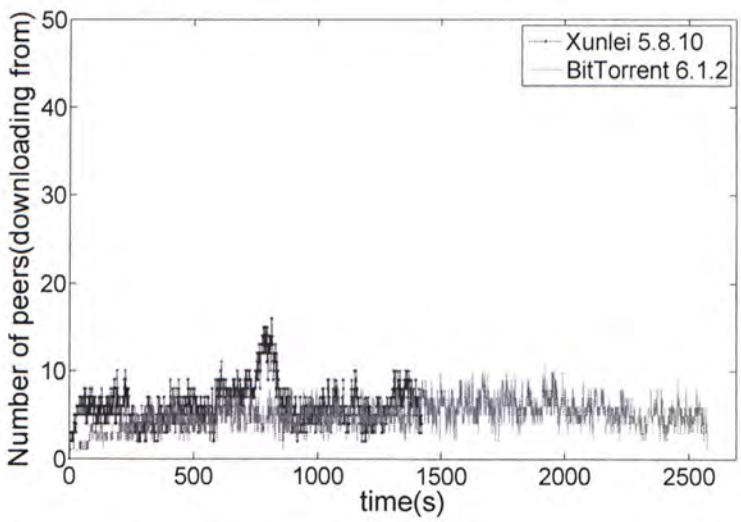
Another possible reason for Xunlei's high downloading speed is its multi-protocol downloading strategy. Running a Xunlei client is like running two clients respectively: one is getting resources from the network where the original file is located, the other one is getting resources from the Xunlei network. Intu-



(a) torrent i, average download speed: Xunlei321.54kB/s; BitTorrent193kB/s.



(b) torrent j, average download speed: Xunlei331.1kB/s; BitTorrent99.9kB/s.



(c) torrent k, average download speed: Xunlei 116.48kB/s; BitTorrent 64kB/s.

Figure 5.14: The evolvement of number of peers a Xunlei/BT client downloads resources from

itively, a Xunlei client can connect to more swarm peers, and hence it has more sources to get the file, which ensures its high downloading speed.

We performed several experiments to evaluate our intuition. We use a Xunlei client and a BitTorrent client to download the file contained in the same torrent file respectively. To make a fair comparison, both these two clients are installed in the same PC and the experiments are performed one immediately after another. We show our results, the number of active swarm peers that upload file chunks to our clients, in Fig. 5.14.

According to previous measurement results about file downloading speed (see Table. 1.1 and Fig. 5.13), Xunlei and BitTorrent can both be the faster one. But usually Xunlei is faster. Consistent with our intuition, one of the reasons seems to be the number of altruistic (partially) seeding neighbors as indicated by Fig. 4.14(a) and Fig. 4.14(b): comparing to BitTorrent clients, Xunlei clients usually download file chunks from more swarm peers and also have higher downloading speed. Indeed there can be various reasons for Xunlei's fast downloading speed, especially when we focus on a particular downloading task instead of numerous ones. Simply the tracker returning some peers with large upload capacity can be one of the reasons. Fig. 4.14(c) can be seen as an example: in this downloading session, the Xunlei client only has active swarm peers with the number slightly larger then that of the BitTorrent client has, but it reaches a downloading speed almost twice as much as the BitTorrent client does.

While Xunlei's multi-protocol strategy provides its clients high downloading speed, it also puts a high upload burden on them, since in P2P networks total download capacity is approximately the same as the total upload capacity. That is why in Table. 1.1, besides downloading speed, the Xunlei client also has higher uploading load.

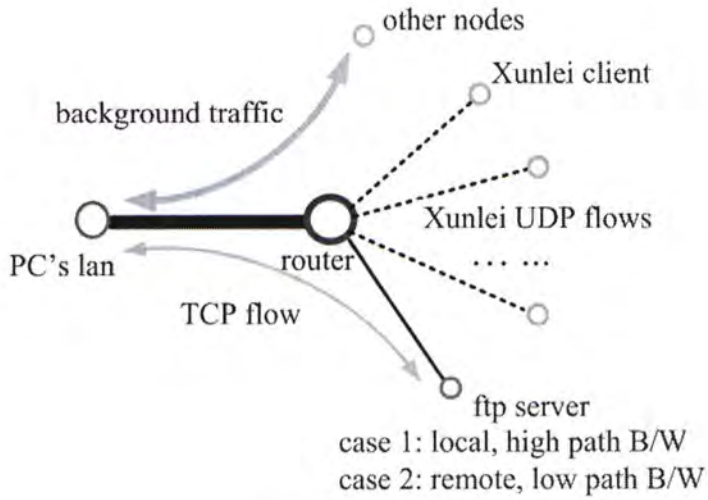
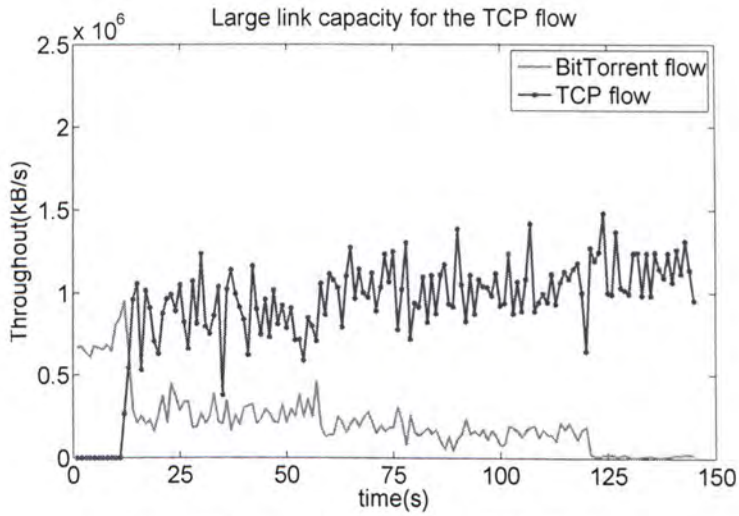


Figure 5.15: Experiment setup

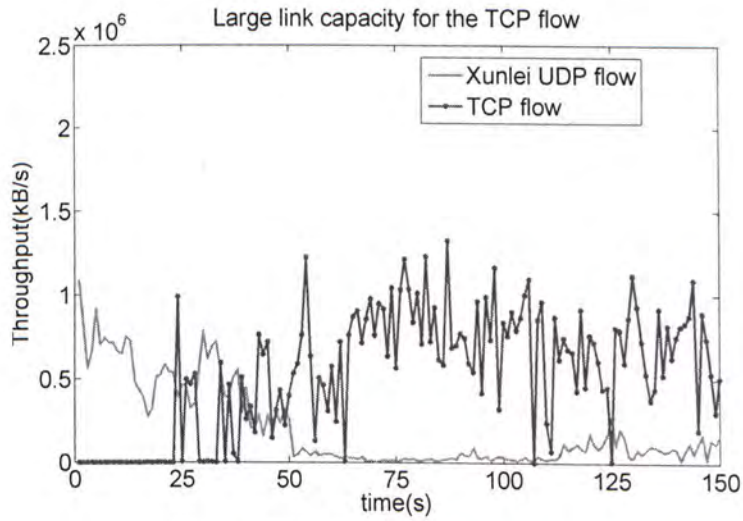
5.3 UDP-based data transmission

Another possible reason is due to Xunlei's packet transmission strategy: how aggressive is Xunlei's flow and congestion control relative to TCP. Just from observing a sequence of Xunlei packets, it is not clear at all if Xunlei implements any window or rate based control for packet transmission. It appears Xunlei simply pushes packets out continuously until packet loses are detected by triple-duplicate-ack or timeout. In case of loses, Xunlei retransmits the (presumed) lost packets, and it is not clear the exact condition under which it resumes new data packet transmissions.

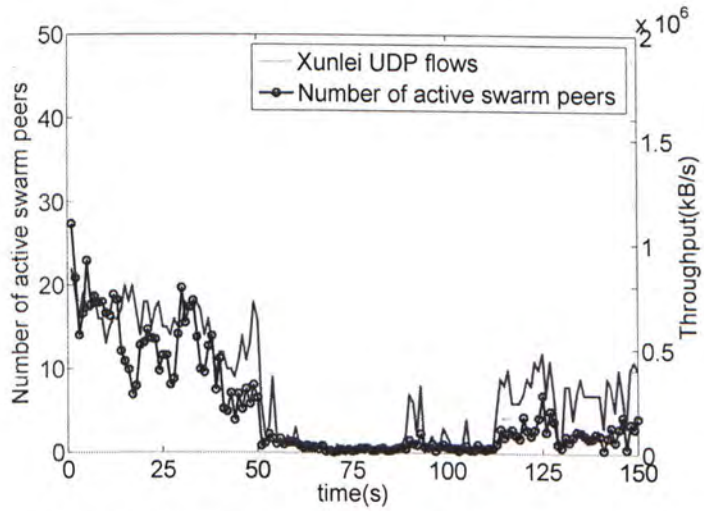
What we can do is to experiment with scenarios with simultaneous Xunlei and TCP flows, and observe how each kind of traffic get affected by the other. Ideally, we would like to do controlled experiments with Xunlei flow(s) and a TCP flow traversing the same path, and compare the result to a Xunlei flow or a TCP flow alone. Unfortunately, it is not easy to control the path Xunlei flows traverse, so we cannot get the TCP flow to share exactly the same path. Also, we cannot avoid possible background traffic.



(a) BitTorrent flows and the TCP flow



(b) Xunlei UDP flows and the TCP flow



(c) Xunlei UDP flows and number of active swarm peers

Figure 5.16: Xunlei's influence: large link capacity for the comparing TCP flow

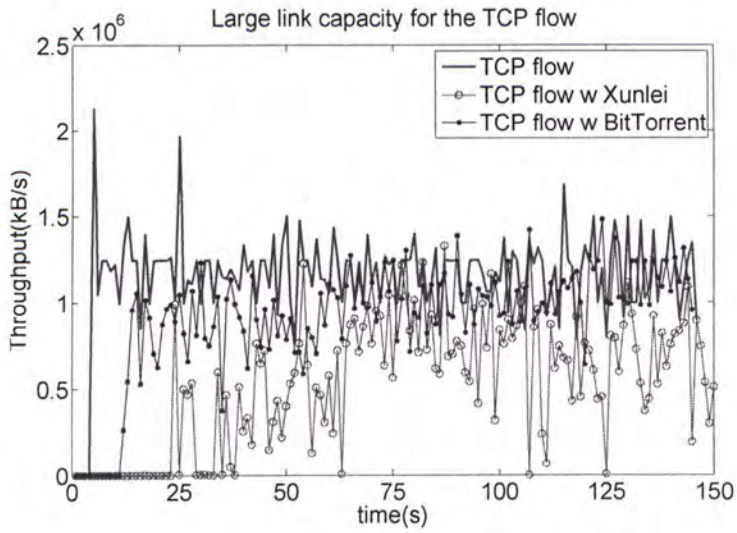
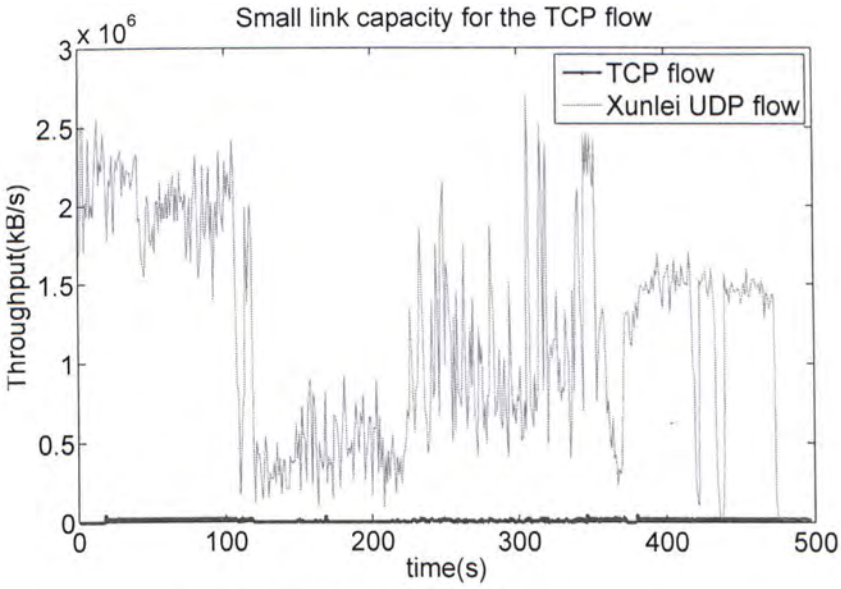


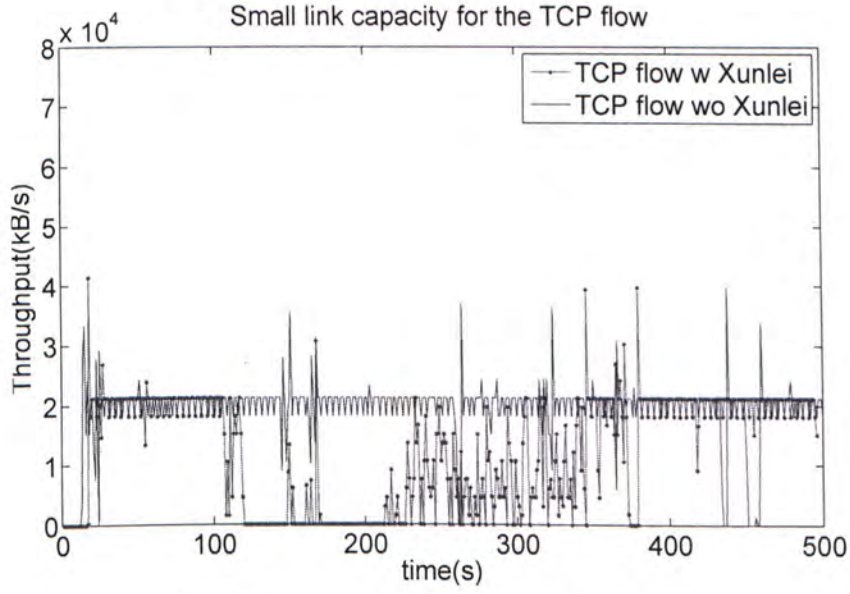
Figure 5.17: Comparison of TCP flow's throughput: large link capacity for the comparing TCP flow

Instead, we pre-arrange two TCP flows. The source of the TCP flows is the same as Xunlei's flows, on the host where Xunlei is running. One of the TCP flow is destined for another host on campus; the other TCP flow is destined for a host far away (outside of campus). In other words, the local TCP flow traverses a high bandwidth path with a small number of hops; whereas the remote TCP flow traverses a long path with lower bandwidth. The situation is depicted in Fig. 5.15. Note, there is also likely to be some background traffic that we cannot control. To minimize the effect of the randomness of the background traffic, we perform the experiments several times to ensure we report a consistent effect. The results are shown in Fig. 5.16, Fig. 5.17 and Fig. 5.18.

For the case with a local TCP flow (traversing a path of high bandwidth), the TCP flow achieves higher bandwidth than either simultaneous BT flows, or simultaneous Xunlei flows, as shown in the first and second sub-figures of Fig. 5.16. But the co-existing Xunlei and TCP flows seem to be sightly worse-off compared to the co-existing BT and TCP flows. We speculate



(a) Xunlei UDP flows and the TCP flow



(b) Comparison of TCP flow's throughput

Figure 5.18: Xunlei's influence: small link capacity for the comparing TCP flow

that the reason for TCP being worse off is due to the aggressiveness of Xunlei in competing with the TCP flow for bandwidth. The reason for Xunlei to be also worse off seems to be due to the few neighbors it is able to get, as shown in the third sub-figure of Fig. 5.16. Figure 5.17 compares the throughput of the TCP flow with (a) no P2P flows, (b) with BT flows, and (c) with Xunlei flows. The relative throughput is (a) better than (b) and (b) better than (c).

For the case with a remote TCP flow (traversing a longer path with lower bandwidth), the TCP flow achieves much less throughput than the simultaneous Xunlei flows. In this case, the instantaneous TCP rates also tends to fluctuate wildly, see Fig. 4.18(b). Due to the higher aggregate rate of Xunlei flows, the TCP flow has to respond more often to congestion signals, leading to much lower throughput.

In the second sub-figure of Fig. 4.18(b), we compare the instantaneous rate of the TCP flow with and without the Xunlei flows, and it is clear that the latter case is much less stable.

□ **End of chapter.**

Chapter 5

Conclusion

In this thesis, we studied peer-to-peer systems in two aspects - we provided an analysis, as well as an evaluation of the design of trackers in P2P systems, and we studied a large-scale P2P file sharing system (Xunlei).

The inherent distributed property of P2P systems require a participating peer to find more swarm peers to get a better performance. The "tracker" function is hence a key enabler for all P2P systems. Traditional server-based trackers hold bottleneck and do not support well the scalability either in the number of objects (e.g. files), or in the number of simultaneous participating peers. DHT-based trackers, however, are much more scalable. Our analysis shows that, the system reliability of DHT-based tracker design can be quite stable in the face of heterogeneous system population and resource popularity, owning to they distributing the workload of searching information to normal peers, and achieving another level of "distributed manner" - P2P systems with DHT-based trackers are distributed not only in distributing contents, but also in searching information (e.g. file info and peer info). Further, our analysis of DHT-based tracker design shows that, among different system and network parameters, peer's average lifetime and the DHT stabilization interval influence its performance (reliability) most.

Having attracted a vast user base (purportedly over 200 mil-

lion users), and achieved very good downloading performance compared to Bittorrent, Xunlei is a system worth closer examination. Yet, being proprietary means it is not easy to uncover its design details. In this thesis, we tried to study Xunlei the best we can using a black-box approach. We find that its multi-protocol design, and the ability to tap into different networks simultaneously is probably the most important reason for its good performance. We believe our measurements and detective work is useful for other colleagues working on P2P algorithms as well.

□ **End of chapter.**

Bibliography

[1] http://baike.baidu.com/view/1506350.htm.

[2] http://bamboo-dht.org/.

[3] http://en.wikipedia.org/wiki/bittorrent_client.

[4] http://en.wikipedia.org/wiki/bittorrent_protocol.

[5] http://en.wikipedia.org/wiki/edonkey_network.

[6] http://en.wikipedia.org/wiki/kad_network.

[7] http://en.wikipedia.org/wiki/selective_repeat_arq.

[8] http://en.wikipedia.org/wiki/xunlei.

[9] http://kankan.xunlei.com/.

[10] http://nsnam.isi.edu/nsnam/index.php/main_page.

[11] http://pdos.csail.mit.edu/p2psim/.

[12] http://www.emule-project.net/.

[13] http://www.gougou.com/.

[14] http://www.opnet.com/.

[15] http://www.wireshark.org/.

[16] http://www.xunlei.com/.

[17] D. Bickson and Y. Kulbak. emule protocol specification. *Master thesis, The Hebrew University of Jerusalem, Jerusalem*, January 2005.

[18] F. E. Bustamante and Y. Qiao. Friendships that last: Peer lifespan and its role in P2P protocols. In *Proceedings of International Workshop on Web Content Caching and Distribution*, Sep. 2003.

[19] D. Carra and E. Biersack. Building a reliable P2P system out of unreliable P2P clients: The case of Kad. In *Proceedings of CoNext'07*, New York, USA, 2007.

[20] B. Cohen. Incentives build robustness in BitTorrent. In *P2P Econornics Workshop*, Berkeley, CA, 2003.

[21] B. Cohen. The bittorrent protocol specification 3. *http://www.bittorrent.org/beps/bep_003.html*, January 2008.

[22] Y. Gu and R. L.Grossman. UDT: UDP-based data transfer for high-speed wide area networks. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 51(7):1777–1799, May 2007.

[23] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of dht routing geometry on resilience and proximity. In *Proceedings of ACM SIGCOMM'03*, Karlsruhe, Germany, August 2003.

[24] Y. Huang, T. Fu, D. Chiu, J. Lui, and C. Huang. Challenges, design and analysis of a large-scale P2P VoD system. In *Proceedings of ACM SIGCOMM'08*, August 2008.

[25] A. L. Jia and D. M. Chiu. Designs and evaluation of a tracker in P2P networks. In *the 8th International Con-*

ference on Peer-to-Peer Computing 2008 (P2P'08), pages 227–230, 2008.

[26] W. Kun and D. H. et al. NDP2PSim: a NS2-based platform for peer-to-peer networks simulations. In *ISPA Workshops 2005*, pages 520–529, 2005.

[27] D. Leonard, V. Rai, and D. Loguinov. On lifetime-based node failure and stochastic resilience of decentralized peer-to-peer networks. In *Proceedings of ACM SIGMETRICS'05*, pages 26–37, Jun. 2005.

[28] J. Li, J. Stribling, R. Morris, M. Kaashoek, and T. M.Gil. A performance vs. cost framework for evaluating DHT design tradeoffs under churn. In *Proceedings of IEEE INFOCOM'05*, pages 225–236, 2005.

[29] A. Loewenstern. The bittorrent protocol specification 5. *http://www.bittorrent.org/beps/bep_005.html*, January 2008.

[30] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 53–65, March 2002.

[31] S. Naichen and B. el al. The state of peer-to-peer simulators and simulations. *ACM SIGCOMM Computer Communication Review*, 37(2), April 2007.

[32] N. Prabhu. *Stochastic processes: basic theory and its applications.* New York: Macmillan, 1965.

[33] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350.

[34] S.A.Baset and H.G.Schulzrinne. An analysis of the Skype peer-to-peer internet telephony protocol. In *Proceedings of IEEE INFOCOM'06*, April 2006.

[35] M. Steiner, T. En-Najjary, and E. W. Biersack. Long term study of peer behavior in the KAD DHT. *IEEE/ACM Transactions on Networking*, May 2009.

[36] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of ACM SIGCOMM'01*, San Diego, California, USA, August 2001.

[37] D. Stutzbach and R. Rejaie. Understanding churn in peer-to-peer networks. In *Proceedings of IMC'06*, October 2006.

[38] G. Tan and S. A. Jarvis. Stochastiv analysis and improvement of the reliability of DHT-based multicast. In *Proceedings of IEEE INFOCOM'07*, 2007.

[39] Z. Yao and D. Loguinov. Understanding disconnection and stabilization of Chord. In *Proceedings of IEEE INFOCOM'08*, pages 1049–1057, 2008.

[40] W. Yiu, X. Jin, and S. Chan. Vmesh: Distributed segement storage for peer-to-peer interactive video streaming. *IEEE Journal on Selected Areas in Communications*, 25(9), Sepetember 2007.

[41] M. Zhang, C. Chen, and N. Brownlee. A measurement-based study of Xunlei. In *Proceedings of PAM'09*, April 2009.

[42] B. Y. Zhao and e. a. John Kubiatowicz. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. In *UC Berkeley Technical Report*, 2001.

[43] S. Zoels, S. Schubert, W. Kellerer, and Z. Despotovic. Hybrid DHT design of mobile environments. In *Proceedings of AP2PC'06*, pages 19–30, 2006.