Structures et Algorithmes pour la coopération pair-à-pair

Moritz Steiner

Structures and Algorithms for Peer-to-Peer Cooperation

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
der Universität Mannheim

Thèse présentée
pour obtenir le grade de docteur
de l´Ecole nationale supérieure des télécommunications

Moritz Steiner

Defense date: December, the 8th 2008

Directeurs de Thèse / Doktorväter:
Professor Dr. Ernst Biersack, EURECOM
Professor Dr. Wolfgang Effelsberg, Universität Mannheim

Abstract

Peer-to-peer overlay networks are distributed systems, without any hierarchical organization or centralized control. Peers form self-organizing overlay networks that are on top of the Internet.

Both parts of this thesis deal with peer-to-peer overlay networks, the first part with unstructured ones used to build a large scale Networked Virtual Environment. The second part gives insights on how the users of a real life structured peer-to-peer network behave, and how well the proposed algorithms for publishing and retrieving data work. Moreover we analyze the security (holes) in such a system.

Networked virtual environments (NVEs), also known as distributed virtual environments, are computer-generated, synthetic worlds that allow simultaneous interactions of multiple participants. Many efforts have been made to allow people to interact in realistic virtual environments, resulting in the recent boom of Massively Multiplayer Online Games.

In the first part of the thesis, we present a complete study of an augmented Delaunay-based overlay for peer-to-peer shared virtual worlds. We design an overlay network matching the Delaunay triangulation of the participating peers in a generalized d-dimensional space. Especially, we describe the self-organizing algorithms for peer insertion and deletion.

To reduce the delay penalty of overlay routing, we propose to augment each node of the Delaunay-based overlay with a limited number of carefully selected *shortcut* links creating a small-world. We show that a small number of shortcuts is sufficient to significantly decrease the delay of routing in the space.

We present a distributed algorithm for the clustering of peers. The algorithm is dynamic in the sense that whenever a peer joins or leaves the NVE, the clustering will be adapted if necessary by either splitting a cluster or merging clusters. The main idea of the algorithm is to classify links between adjacent peers into short intra-cluster and long inter-cluster links.

In a structured system, the neighbor relationship between peers and data locations is strictly defined. Searching in such systems is therefore determined by the particular network architecture. Among the strictly structured systems, some implement a distributed hash table (DHT) using different data structures. DHTs have been actively studied in the literature and many different proposals have been made on how to

organize peers in a DHT. However, very few DHTs have been implemented in real systems and deployed on a large scale. One exception is KAD, a DHT based on Kademlia, which is part of eDonkey, a peer-to-peer file sharing system with several million simultaneous users.

In the second part of this thesis we give a detailed background on KAD, the organization of the peers, the search and the publish operations, and we describe our measurement methodology. We have been crawling KAD continuously for more than a year. We obtained information about geographical distribution of peers, session times, peer availability, and peer lifetime. We found that session times are Weibull distributed and show how this information can be exploited to make the publishing mechanism much more efficient.

As we have been studying KAD over the course of the last two years we have been both, fascinated and frightened by the possibilities KAD offers. We show that mounting a Sybil attack is very easy in KAD and allows to compromise the privacy of KAD users, to compromise the correct operation of the key lookup and to mount distributed denial-of-service attacks with very little resources.

Zusammenfassung

Peer-to-peer Overlay Netwerke sind verteilte Systeme ohne jede hierarchische Organisation oder zentrale Kontrolle. Die Teilnehmer bilden auf Anwendungsebene sich selbst organisierende Overlay-Netzwerke, die über das unterliegende Netzwerk, das Internet, paarweise miteinander Verbindungen aufbauen können.

Beide Teile dieser Dissertation behandeln Peer-to-Peer Overlay-Netze. Der erste Teil beschäftigt sich mit unstrukturierten Overlays, die unter anderem benutzt werden können, um virtuelle Welten im großen Maßstab aufzubauen. Der zweite Teil gewährt Einblicke darüber, wie sich die Benutzer eines real existierenden strukturierten Peer-to-Peer Netzwerkes verhalten und wie gut die vorgeschlagenen Algorithmen für die Publikation und Suche von Daten funktionieren. Darüber hinaus wird die Sicherheit eines solchen Systems analysiert.

Networked virtual environments, auch bekannt als verteilte virtuelle Umgebungen, sind Computer-generierte synthetische Welten, die es mehreren Teilnehmern ermöglichen, gleichzeitig zu agieren. Es wurden viele Anstrengungen unternommen, um Nutzern die Interaktion in möglichst realistischen virtuellen Umgebungen zu ermöglichen. Dadurch wurde der Boom von Massively Multiplayer Online Games in den letzten Jahren erst ermöglicht.

Im ersten Teil dieser Dissertation stellen wir eine komplette Studie eines Delaunay-basierten Peer-to-Peer Overlays für verteilte virtuelle Welten vor. Wir entwerfen ein Overlay-Netz, das mit der Delaunay-Triangulierung der teilnehmenden Peers in einem d-dimensionalem Raum übereinstimmt. Vor allem beschreiben wir die sich selbst organisierenden Algorithmen für das Einfügen und Entfernen eines Peers.

Um die erhöhte Laufzeit, die durch das Routing im Overlay entsteht, zu reduzieren, schlagen wir vor, jeden Knoten im Delaunay-basierten Overlay mit einigen sorgfältig ausgewählten *Abkürzungen* anzureichern, so dass eine sogenannte *small-world* entsteht. Anschließend zeigen wir, dass eine kleine Anzahl von Abkürzungen ausreichend ist, um die Laufzeit einer Nachricht im Overlay signifikant zu reduzieren.

Wir präsentieren einen verteilten Algorithmus zum Clustern von Peers. Der Algorithmus ist adaptativ in dem Sinne, dass jedesmal, wenn ein Peer dem NVE beitritt oder es verlässt, das Clustering wenn nötig angepasst wird, indem ein Cluster aufgeteilt wird oder Cluster zusammengeschlossen werden. Die zentrale Idee

des Algorithmus ist es, die Verbindungen zwischen benachbarten Peers in kurze intra-cluster- und lange inter-cluster-Verbindungen aufzuteilen.

In einem strukturierten Peer-to-Peer System ist die Nachbarschaftsbeziehung zwischen Peers sowie der Ort für die Datenspeicherung strikt festgelegt. Die Suche in einem solchen System ist daher durch die spezielle Netzwerkarchitektur bestimmt. Unter den strukturierten Systemen implementieren einige eine verteilte Hash-Tabelle (Distributed Hash Table, DHT). DHTs sind in der Literatur ausführlich untersucht worden, und es sind viele verschiedene Vorschläge für die Organisation der Peers gemacht worden. Dennoch sind nur sehr wenige DHTs in einem realen System implementiert worden und im großen Maßstab zur Anwendung gekommen. Eine Ausnahme ist KAD, eine auf Kademlia basierende DHT, die Teil von eDonkey ist, einem Peer-to-Peer Netz, das von mehreren Millionen Nutzern gleichzeitig benutzt wird.

Im zweiten Teil dieser Dissertation stellen wir die Funktionsweise von KAD vor, seine Organisation der Peers, sowie seine Publikations- und Suchoperation. Um ein System dieser Größenordnung zu untersuchen, haben wir unsere eigene, skalierbare und robuste, Meßmethodologie entwickelt. Auf diese Weise war es uns möglich, KAD ohne Unterbrechungen während eines gesamten Jahres zu vermessen und eine Vielzahl von Informationen zu erhalten. So haben wir Daten über die geographische Verteilung, die Sitzungszeiten, die Verfügbarkeit, sowie die Lebenszeit der Peers gewonnen. Ein besonders interessantes Ergebnis ist, dass die Sitzungszeiten Weibull-verteilt sind. Wir zeigen, wie diese Information dazu genutzt werden kann, den Publikationsmechanismus effizienter zu gestalten.

Während wir in den vergangenen zwei Jahren mit den Funktionsweisen von KAD immer vertrauter wurden, haben uns die Möglichkeiten, die KAD bietet, sowohl fasziniert als auch erschreckt. KAD erlaubt es einem einzelnen Benutzer ohne große Mühe viele verschiedene Identitäten anzunehmen. Diese können dazu benutzt werden spezifische Daten über Nutzer zu sammeln, die korrekte Funktionsweise der Suche zu unterbinden und verteilte Denial-of-Service Angriffe mit sehr wenigen Ressourcen durchzuführen.

Acknowledgements

This thesis was done in a "co-tutelle" way. This means I had two advisors, two institutions, and two sets of rules. You may ask if the additional effort that comes along is worth the benefit. The answer is a clear yes. I had the opportunity to see and compare two different institutes, two different countries, two different administrations, and two different IT departments: two different ways of conducting research. This is a great experience.

Most important were of course my two advisors: Prof. Biersack at Eurécom and Prof. Effelsberg at the Universität of Mannheim. In a talk of Jim Kurose on how to do a good and successful Ph.D. the first point is: "Choose a good advisor!". I did even better, I chose two of them. In German we use the word *Doktorvater* for Ph.D. advisor, which in my opinion expresses in a much more complete way the role a Ph.D. advisor plays in the best case. As all fathers are different, my two *Doktorväter* differ, too. One leaves more scope and supervises from a further distance, the other one prefers a closer collaboration. Both of them taught me how to do research. One helped me not to miss the big picture, the other one taught me all the necessary methods and rigor to conduct a study.

Luckily, my two advisors know each other very well. Before beginning my Ph.D., I already had written my master thesis with the same advisors. I knew therefore that it would not be a mistake to choose them again. In fact Prof. Effelsberg, who had already sent me as an Erasmus student to the university of Nice, asked me if I wanted to return to southern France for my master thesis, where a colleague of him works in a small but nice research lab. Before I could even think about the proposal and send an application, Prof. Biersack had already called me. He was already a step ahead of me. He not only wanted me for a master thesis but he wanted me to continue later towards my Ph.D. This was of course too fast for me, but eventually, after having finished my master studies, I was happy to continue.

Thus, I had not only the advantage of having two advisors but I was also part of two research teams. I'm very grateful for everything I learned from my colleagues that was useful for the daily life of a Ph.D. student. Special thanks to the *alten Hasen* in Mannheim who helped me with Linux, LaTeX , Perl, etc. When I came to Eurécom after the time in Mannheim, I was already a "senior" Ph.D. student myself. I hope I could pass on some of my practical experience to new Ph.D. students as well as my know-how acquired during the first 18 months of my thesis work. The atmosphere in the *Networking Department* at Eurécom as well as at the *Lehrstuhl für Praktische*

Informatik IV was very friendly and I simply want to thank all the colleagues who accompanied me during my thesis.

There are some colleagues I want to mention especially, because I've been working closely together with them and I gained a lot from their expertise. The design and clear description of the algorithms for the n-dimensional distributed Delaunay Triangulation were also the work of Gwendal Simon, France Télécom R&D at that time. Thorsten Holz, Universität Mannheim, contacted me to ask for some help regarding the Storm Worm that uses a peer-to-peer network for communication. The evaluation of the KAD crawl data would not have been possible without the statistical skills of my office mate at Eurécom Taoufik En-Najjary. Damiano Carra was not only a house mate of mine and a colleague at Eurécom but also worked with me on the evaluation of the content retrieval process in KAD.

Last but not least I want to mention the computing center in Mannheim. Without its extremely reliable service, without the abondant resources it provides, and without the patience of its staff – more than once my experiments triggered their intrusion detection system – most of the data I collected could not have been collected and most of the papers I was able to publish, would never have been written. At Eurécom I want to thank Patrick Petitmengin of the Service Informatique, who was always available in case of emergency and who assembled and installed the measurement machines in the lab.

I want to thank my parents for giving me the freedom to make my own decisions, but at the same time always supporting me once the decision has been taken.

Contents

List of Figures

List of Tables

CHAPTER 1
Introduction

"Take young researchers, put them together in virtual seclusion, give them an unprecedented degree of freedom and turn up the pressure by fostering competitiveness."

– James D. Watson –

1.1 Peer-to-peer networks

Peer-to-peer overlay networks are distributed systems without any centralized control. Peers form a self-organizing overlay network that runs on top of the Internet. These overlays offer various features such as robust routing, efficient search of data items, selection of nearby peers, redundant storage, anonymity, scalability and fault tolerance. A peer-to-peer network does not have the notion of clients or servers but only equal peer nodes that simultaneously function as both "clients" and "servers" to the other nodes of the network.

New challenges such as topology maintenance arise, as nodes can join or leave at any time, and efficient content search, as no node has a complete knowledge of the entire topology. Peer-to-peer overlay networks do not arise from the collaboration between established and connected groups of systems, and without a reliable set of resources to share. The basic units are commodity PCs instead of well-provisioned facilities.

The peers have autonomy, i.e., they are able to decide about services they wish to offer to other peers. Peers are assumed to have temporary network addresses, and have to be recognized and reachable even if their network address has changed. That is why issues of scale and redundancy become much more important than in traditional (centralized or distributed) systems.

The peers cannot necessarily trust each other because of their autonomy. Indeed attacks in peer-to-peer networks are easy to set up. It is possible to join the net-

work with multiple identities, the so called *sybils*, and affect the routing of messages, and the publication and lookup of content. This way content can be made invisible, *eclipsed*, for benign participants. Moreover content can be polluted by bogus publications.

Peer-to-peer networks have three main principles, which specify a fully-distributed, cooperative network design with peers building a self-organizing system:

- The principle of **sharing resources**: all peer-to-peer systems involve an aspect of resource sharing, where resources can be physical resources, such as disk space or network bandwidth, as well as logical resources, such as services or different forms of knowledge. By sharing of resources, applications can be realized that could not be set up by a single node. They scale with the number of participants.
- The principle of **decentralization**: this is an immediate consequence of sharing of resources. Parts of the system or even the whole system are no longer operated centrally. Decentralization is in particular interesting in order to avoid single points of failure or performance bottlenecks in the system.
- The principle of **self-organization**: when a peer-to-peer system becomes fully decentralized there exists no longer a node that can centrally coordinate its activities, or a central database to store global information about the system. Nodes have to self-organize themselves, based on whatever local information is available and by interacting with locally reachable nodes (neighbors). The global behavior then emerges as the result of all the local behaviors.

File Sharing, Video Streaming, Telephony, Backup, Games	P2P Application Layer
Meta-Data (File Management), Messaging, Scheduling	P2P Services Layer
Node State, Keep Alive, Lookup, Join, Leave	Overlay Node Management
TCP and UDP over IP	Transport Layer

Table 1.1: The Peer-to-Peer Protocol Stack.

Table 1.1 shows an abstract peer-to-peer overlay architecture, illustrating the components in the overlay communications framework. The Transport layer is responsible for the connection of desktop machines over the Internet. The dynamic nature of peers implies that peers can join and leave the peer-to-peer network at any time. This is a significant problem for peer-to-peer systems, as they must maintain consistent information about peers in the system in order to operate most effectively. All the layers above the Network layer are situated at the *application layer* of the OSI model. Thus a peer-to-peer system can be viewed as an application-level Internet on top of the Internet.

The Overlay Nodes Management layer covers the management of peers, which include the operations to join and leave the network, the discovery of peers, and routing algorithms. The neighbors of a peer are chosen depending on the look-up algorithm used. So we can say that the look-up algorithm specifies or defines a peer-to-peer network.

The peer-to-peer Services layer supports the underlying peer-to-peer infrastructure through scheduling of tasks, content and file management. Meta-data describe the content stored across the peers and the location information.

The peer-to-peer Application layer is concerned with tools and applications that are implemented with specific functionalities on top of the underlying peer-to-peer overlay infrastructure. File sharing was the first and oldest application for peer-to-peer networks. Today it has been complemented by a wide variety of other applications using a peer-to-peer infrastructure, such as telephony (e.g., Skype), video streaming (e.g., Zattoo), distributed backup (e.g., Wuala), and games.

The peer-to-peer overlay network consists of all the participating peers as network nodes. Every peer has a list of other peers he knows. They are used to forward any kind of messages, for routing, publishing, or searching purposes. There are links between any two peers that know each other: i.e., if a participating peer knows the location of another peer in the network, then there is a directed edge from the former node to the latter in the overlay network. Based on how the nodes in the overlay network are linked to each other, we can classify them as *unstructured* or *structured*.

Unstructured Peer-to-Peer Networks An unstructured peer-to-peer network is formed when the overlay links are established arbitrarily. A new peer that wants to join the network can copy existing links of another node and then form its own links over time. The network uses flooding as the mechanism to send queries across the overlay, with a limited scope. When a peer receives the flood query, it sends back a list of all the content matching the query to the originating peer.

While flooding-based techniques are effective for locating highly replicated items and are resilient to peers joining and leaving the system, they are poorly suited for locating rare items. Clearly this approach is not scalable as the load on each peer grows linearly with the total number of queries and the system size. Thus, unstructured peer-to-peer networks face one basic problem: peers become overloaded, therefore, the system does not scale when handling a high rate of aggregate queries and sudden increase in system size.

Structured Peer-to-Peer Networks In a structured peer-to-peer system, the neighbor relationship between peers and data locations is strictly defined. Among the structured systems, some implement a distributed hash table (DHT) using different data structures. The structure of a DHT can be decomposed into several main components. The foundation is an abstract key space, which typically consists of large inter values, e.g. the range from 0 to $2^{128} - 1$. A random unique key (identifier) from this keyspace is assigned to each participant. Each node stores a partial view of the whole distributed system, i.e., it maintains a small routing table consisting of its neighboring peers. Together these links form the overlay network. Based on this information, the routing procedure traverses several nodes, approaching the destination with each hop until the destination node is reached. This style of routing is sometimes called key based routing.

A data item is mapped to the same keyspace as the participating peers by hashing either the filename or the binary file itself using a consistent hash function. Thus, DHTs implement a proactive strategy to retrieve data by structuring the search space in a deterministic way. The way content is assigned to one or more participant(s) differs among the proposed DHTs. This mapping scheme of the data items to the peers defines the routing strategy on how to find the node who stores the data item. The routing is used in the same way for both basic functions: publish and lookup. In most implementations, this will not be a single node, but every data item is replicated several times on different nodes who are close to the data item in the keyspace.

In theory, DHT-based systems can guarantee that any data object can be located on average in $O(\log N)$ number of overlay hops, where N is the number of peers in the system. The underlying network path between two peers can be significantly different from the path on the DHT-based overlay network. Therefore, the lookup latency in DHT-based peer-to-peer overlay networks can be quite high and could adversely affect the performance of the applications running on top.

Although structured peer-to-peer networks can efficiently locate rare items since the key-based routing is scalable, they incur significantly higher overheads than unstructured peer-to-peer networks for popular content (cf. Section 10.4).

1.2 Networked virtual environments

Networked virtual environments (NVEs) [53, 62], also known as distributed virtual environments, are computer-generated, synthetic worlds that allow simultaneous interactions of multiple participants. From the early days of SIMNET [17], a joint project of the U.S. Army and Defense Advanced Research Project Agency (DARPA) between 1983 and 1990 for large scale combat simulations, to the recent boom of *Massively Multiplayer Online Games* (MMOG), many efforts have been made to allow people to interact in realistic virtual environments.

Most of the existing MMOGs are role-playing games; first-person shooter games or real-time strategy games are usually divided into many small isolated game sessions with a handful of players each. As of early 2006, the most popular title, World of Warcraft, has attracted more than six million subscribers within two years, while the game Lineage also has more than four million subscribers worldwide. More than 100,000 peak concurrent users are frequently reported for major MMOGs, making scalability a defining trait.

Works of science fiction, such as Neal Stephenson's novel "Snow Crash" [118] and the Matrix movies, give an impression of what a 3D environment that is truly consistent, persistent, realistic and immersive could be like. Progression in technology, converging advances in CPU, 3D acceleration and bandwidth may make the vision come true in the near future.

NVE scalability is fundamentally a resource issue, and is determined by whether the system has enough resources to continuously accommodate new participants. However, existing server-based architectures pose inherent scalability limits and significant costs. Though much effort has been spent on improving scalability, no real system exists as yet that could support more than one million concurrent users. The popularity and rapid growth of large-scale NVEs such as MMOGs will likely pose serious challenges to existing networks and architectures in the near future. On the other hand, a new class of peer-to-peer applications that seeks to realize large-scale NVEs has recently appeared as an alternative.

A number of issues are involved in building an NVE system; several are described as follows:

- **Consistency** - For meaningful interactions to happen, all users' perceptions of the virtual world must be consistent. This includes maintaining states and keeping events synchronized. The inherently distributed nature of peer networks makes it difficult to guarantee reliable behavior. The most widespread solution to ensure consistency across NVEs is to keep redundant information in different peers. For example, upon a detection of a failure, the task can be restarted on other available machines. Alternatively, the same task can be initially assigned to multiple peers. In messaging applications lost messages can be resent or sent along multiple paths simultaneously. Finally, in file sharing applications, data can be replicated across many peers. In all serverless peer-to-peer systems, states about neighbors and connections must be kept in different peers to ensure the consistency after the crash of one or more peers.

  Zhou et al. [141] classify NVE event consistency into two main groups:

  - Causal order consistency - Events must happen in the same order as they occur, as humans have deeply-rooted concepts about the logical order of sequences of events.

  - Time-space consistency - In an NVE system, messages are sent to notify others about position updates. However, due to network delay and clock asynchrony among computers, it is possible for hosts to receive updates with different delays and thus interpret the order of events differently. Inconsistencies therefore could occur for entity positions at a given logical time.

  Note that causal order consistency and time-space consistency are not necessarily related to each other (i.e., it is possible to preserve causal order consistency but violate time-space consistency). This problem is particularly evident when forecasts are used to compensate missing updates. On the other hand, in p2p networks the concept of consistency generally refers to what may be called topology consistency, which is whether each node in the p2p system holds consistent views of the parts of the network they share (note that each node only maintains a local view of the complete topology).

- **Scalability** is usually concerned with the number of simultaneous users in a NVE. One important challenge is to allow all people to interact in the same environment. This is achieved by developing new systems that do not rely on a centralized server. Thus the server can not be overloaded and crash due to to increasing usage. No investment is needed for servers. The nodes only know the nodes in their *attention radius*. While the virtual world is infinite, in this radius the number of peers is limited. A trade-off has to be made between knowing enough neighbors to interact and not keeping connections open to too many neighbors.

- **Reliability** and fault resilience is important to make an NVE a service with quality. To ensure this point peers may need some redundant information to cope with a crash of one or more neighbors. One of the primary design goals of a peer-to-peer system is to avoid a central point of failure. Although most (pure) peer-to-peer systems already do this, they nevertheless are faced with failures commonly associated with systems spanning multiple hosts and networks: disconnections / unreachability, partitions, and node failures. These failures may occur more often in some networks (e.g., wireless) than in others (e.g., wired enterprise networks). In addition to these random failures, personal machines are more vulnerable than servers to security attacks or viruses. It is be desirable to continue active collaboration among the remaining connected peers in the presence of such failures.

- **Performance** - NVEs are simulations of the real world. Performance therefore is important to adapt the virtual world fast enough to allow for the impression of realism and to keep the different views consistent. Because of the decentralized nature of these models, performance is influenced by three types of resources: processing, storage and networking. In particular, networking delays can be significant in wide area networks. Bandwidth is a major factor when a large number of messages is propagated in the network and large amounts of files are being transferred among many peers. This limits the scalability of the system. Performance in this context cannot be measured in milliseconds, but rather tries to answer questions of how long it takes to retrieve a file or how much bandwidth a query will consume. These numbers have a direct impact on the usability of a system.

- **Security** is a big challenge for NVEs, especially if they use a peer-to-peer infrastructure without centralized components. Transforming a standard client into a server poses a number of risks to the system. Only trusted or authenticated peers should have access to information and services provided by a given node. Unfortunately, this requires either potentially painful intervention from the user, or interaction with a trusted third party. Centralizing the task of security is often the only solution even though it voids the p2p benefit of a distributed infrastructure. Another way is to introduce the notion of trust: In the physical world we trust someone who has a good reputation. The concept of reputation can be adopted to the p2p world: you only trust a node you know or a node for which you get a recommendation from a known trusted node.

- **Persistency** - To create sophisticated contents, certain data such as user profiles and valuable virtual objects must be persistently stored and accessible across user sessions. An example of a virtual object may be a bar where people can meet. This information is in most cases stored on a central server to allow the users to log into the virtual world with the same identity from different computers.

We consider scalability to be the most important issue if we plan to build truly massive worlds and applications, which millions of people can participate in and enjoy. Therefore, this thesis focuses on finding a solution for the scalability problem in NVEs. Existing approaches to improve scalability mainly rely on enhancing the server capacity in a client-server architecture. Further scalability is achieved by clustering servers and by dividing the game universe into multiple different, or parallel, worlds and spreading the users over them. However, a client-server architecture has an inherent upper limit in its available resources (i.e., processing power and bandwidth capacity) and is expensive to deploy and to maintain. Distributed systems relying on a peer-to-peer architecture have emerged in recent years as an alternative that promises scalability and affordability. We attempt to apply a peer-to-peer architecture to NVE design, and to address the scalability problem.

1.3 Organisation of the Thesis

In this section we present the main topics developed in this thesis.

Both parts of this thesis deal with peer-to-peer overlay networks, the first part with unstructured ones used to build a large-scale Networked Virtual Environment. The second part gives insights on how the users of a real life structured peer-to-peer network behave, and how well the proposed algorithms for publishing and retrieving data works. Moreover we analyze the security (holes) in such a system.

1.3.1 An Augmented Delaunay Overlay for Decentralized Virtual Worlds

In the first part of the thesis we introduce a set of distributed algorithms with the aim to build a scalable virtual world. To create a large-scale virtual world the traditional client-server model does not scale, and a peer-to-peer based approach is required that constructs an overlay connecting all the participants. NVEs have several requirements that influence the choice of the overlay:

- Peers must be able to freely choose their peerIDs, which in fact reflects their position in the overlay.
- Peers move around, and their positions and therefore peerIDs will change.

- The overlay must efficiently support the communication of a peer with its close-by neighbors.

There exist a large number of structured overlays such as Pastry [100], Tapestry [140], Chord [120], and CAN [98]. However, these overlays typically assign to each peer a fixed peerId, for instance the hash of the peer's IP address, which is not appropriate for NVEs.

One notable difference between P2P-NVE and DHT or file sharing is that the problem of content search is greatly simplified: as each node's area of interest is limited, the desired content is localized and easily identified. This differs from file sharing, where the desired content may potentially be on any node. The content search in peer-to-peer systems thus becomes a neighbor discovery problem.

Delaunay triangulation [29], on the other hand, meets all these requirements. Structured overlays based on Delaunay triangulation have been studied previously [73]. However these works were restricted to the special case of two dimensions only. In Section 3 we propose algorithms for the fully distributed computation and maintenance of the n-dimensional Delaunay Triangulation. A Delaunay-based overlay organizes peers according to their *position* in the NVE. However, as do most overlays, a Delaunay based overlay completely ignores the position of the peers in the physical network. As a consequence, two peers that are neighbors in the overlay may be physically far away, and any message sent over a sequence of overlay hops will experience a significant *delay penalty*.

To reduce the delay penalty of overlay routing, we propose in Section 4 to augment each node of the Delaunay-based overlay with a limited number of carefully selected shortcut links. These shortcuts are chosen in a way that they are short in the underlay but long range in the overlay. Last we introduce in Section 5 a distributed clustering algorithm that allows for the self-organizing clustering of the nodes.

1.3.2 Measurements of real-world peer-to-peer networks

Distributed hash tables (DHTs) have been actively studied in the literature, and many different proposals have been made on how to organize peers in a DHT. However, very few DHTs have been implemented in real systems and deployed on a large scale. One exception is the DHT KAD, an implementation of Kademlia, a peer-to-peer file sharing system with several million simultaneous users.

In Section 8 we give a detailed background on KAD, and we describe our measurement methodology. We present the KAD crawler we have designed and implemented. The speed of this crawler allows us to crawl the entire KAD ID space which has never been done before. We have been crawling KAD continuously for more than a year, the results are presented in Section 9. We obtained information about geographical distribution of peers, session times, peer availability, and peer lifetime. We also evaluated to what extent information about past peer uptime can be used to predict the remaining uptime of the peer. We found that session times are Weibull

distributed and show how this information can be exploited to make the publishing mechanism much more efficient.

Peers are identified by the so-called KAD ID, which was up to now assumed to be persistent. However, we observed that a large number of peers change their KAD ID, sometimes as frequently as after each session. This change of KAD IDs makes it difficult to characterize *end-user* behavior. However, by tracking end-users with static IP addresses, we could measure the rate of change of KAD IDs per end-user and the end-user lifetime.

In Section 10 we present our measurement results on the content shared among users in KAD. Since the measurement technique is based on the Sybil attack, we show how easy it is in KAD to mount such an attack that allows to compromise the privacy of KAD users, to compromise the correct operation of the key lookup and to mount distributed denial-of-service attacks with very little resources. Moreover we present some results of the content pollution attack we ran against the network of the Storm bots that use a Kademlia-based DHT for communication.

We provide distributions of file types, formats, and sizes. In a first evaluation, we notice that publishing new content in a KAD system is much more expensive than searching and retrieving existing content. Indeed, measurements show that of all the Internet traffic generated by KAD-based peer-to-peer networks, 90% is for publishing and 10% for retrieving existing files. Moreover, the most frequently published keywords are meaningless stopwords.

We analyze in detail the content retrieval process of KAD in Section 11. In particular, we present a simple model to evaluate the impact of different design parameters on the overall lookup latency. We then perform extensive measurements on the lookup performance using an instrumented client. From the analysis of the results, we propose an improved scheme that is able to significantly decrease the overall lookup latency without increasing the overhead.

For comparison with the results obtained on KAD we also performed measurements on the DHT of the BitTorrent client Azureus that we present in Section 12. The users come from different regions; the US are dominant whereas they play only a minor role in KAD. The users in Azureus tend to stay online for more than twice the time of the KAD users.

Part I

An Augmented Delaunay Overlay for Decentralized Virtual Worlds

CHAPTER 2
Introduction

"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."
– Leslie Lamport –

Massive and persistent virtual environment which allows millions of people to participate simultaneously may eventually happen on the Internet. There are many technical and architectural issues that need to be resolved before such a true cyberspace can be realized. The primary among these needs is a scalable architecture that handles large numbers of simultaneous users. These worlds will allow people to interact as they do in the real world, e.g., to speak (broadcast audio or video) to people interested in a common topic. Todays massively multi-player online games provide a good impression of what such worlds could be like. It somehow corroborates the concept of a massively shared public virtual world depicted by Neal Stephenson in his science fiction novel *Snow crash* [118].

These applications usually rely on servers, whose owners can monitor the behaviors of inhabitants and ban those who exhibit unexpected characteristics. A second interest for peer-to-peer virtual environments is related to scalability issues. As a dramatic growth of the number of on-line gamers is expected, the stress on central servers increases and advocates for a better distribution of resources involved in the system management. Another motivation for fully decentralized virtual places comes from business perspectives. On-line games are now valuable research venues and platforms where millions of users from a broad age range interact and collaborate on a daily basis [138].

This chapter deals with a peer-to-peer system for a massively distributed virtual world, owned by nobody except inhabitants and populated by a potentially unlimited number of users.

2.1 Networked Virtual Environments

In shared virtual worlds (Networked Virtual Environments), peers are characterized by a position in a three-dimensional space. These worlds are dynamic: peers join and leave the world, move from one virtual place to another, and interact in real-time. The system is intended to provide a similar perception of the same scene for any two peers. This property, called *coherency*, can be obtained when every object – avatar of a player or other virtual object – is aware of all objects within its virtual surroundings and of all events occurring nearby. In a centralized system, a server knows at any time the positions of all peers, so it can easily alert peers about new neighbors and main close events.

Decentralized systems for shared virtual worlds have been studied since the early 90's. Some early applications rely on a two-tier architecture where the virtual space is partitioned into disjoint cells, each cell being under the responsibility of either a server or a super-peer [134, 142]. The partitioning can be dynamically adjusted to cope with variations of virtual densities [68]. Each cell is associated with a multicast address, so moving implies joining and leaving multicast groups when crossing cell borders. Some recent works use a Distributed Hash Table (DHT) to equally share the responsibility of inhabitants for virtual [42, 62], however these applications face latency problems due to message relay. In another approaches, peers carry their *auras*, sub-spaces that bind the presence of an object and act as an enabler of potential interaction [7]. This concept adopted in several applications [95, 130] requires that two peers are alerted when their auras collide, so the problem persists.

A promising approach is based on mutual notification: neighbor discovery is ensured through collaborative notifications between connected nodes. The idea consists of maintaining a logical overlay such that each peer has a direct logical connection with its virtual neighbors and is able to alert two neighbors when they have to be introduced [59]. Ideally, each peer would be connected with all the peers in its aura, these neighbors being able to monitor its aura's boundaries. In this context, an overlay network based on a Delaunay triangulation is appealing [53]. Indeed, as depicted in Figure 2.1, a Delaunay triangulation [29] links two peers when their Voronoi regions share a boundary [5, 88]. A Delaunay triangulation for a set P of points in the plane is a triangulation $DT(P)$ such that no point in P is inside the circumcircle of any triangle in $DT(P)$. Delaunay triangulations maximize the minimum angle of the triangles; they tend to avoid skinny triangles. Generalized to d-dimensions the definition is as follows: For a set P of points in the d-dimensional Euclidean space, a Delaunay triangulation is a triangulation $DT(P)$ such that no point in P is inside the circum-hypersphere of any simplex in $DT(P)$. A Voronoi diagram [132] of a set of peers in an Euclidean space tessellates the whole space. So, any subspace S of a virtual world can be monitored by the peers whose Voronoi region overlaps S. Hence, a peer can be notified of any events in its surroundings subspace if it knows all peers monitoring it [53].

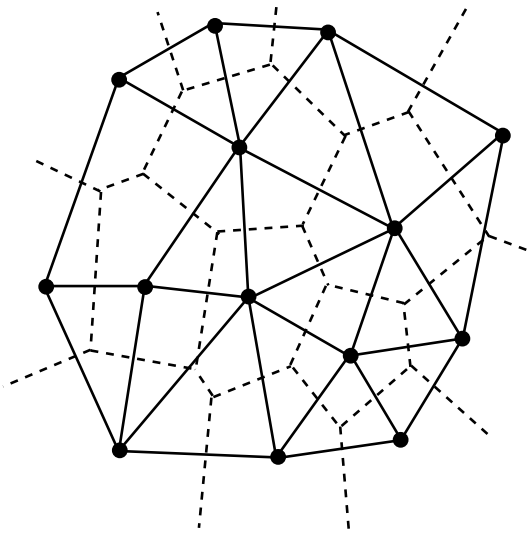


Figure 2.1: Delaunay Triangulation (solid lines) and Voronoi diagram (dashed lines) in two dimensions

2.2 Contributions

In this part of the thesis, we present the design of an augmented Delaunay-based overlay for peer-to-peer massively shared virtual worlds.

Previous proposals of Delaunay-based overlays in dynamic distributed systems rely on an angular feature that is specific to two-dimensional spaces [3, 73]. An overlay based on a Delaunay triangulation has also been studied in ad-hoc networks [72], but some features of wireless protocols ease the detection of neighbors, and the algorithms focus on two-dimensional spaces. These studies can not be applied to higher dimensional spaces. However, realistic representations of virtual worlds require three dimensions. We design an overlay network matching the Delaunay triangulation of the participating peers in a generalized d-dimensional space. Especially, we describe the distributed algorithms for peer insertion and deletion in Section 3.

The resulting overlay organizes peers according to their position in the virtual environment. However, as do most overlays, the position of the peers in the physical network is completely ignored. As a consequence, two neighbors in the overlay may be physically far away, so message exchange along a sequence of overlay hops may experience a significant delay. Yet, entering into the virtual world and teleporting require messages to do a greedy walk of average length equal to $n^{1/d}$ hops. To reduce the delay penalty of overlay routing, we propose in Chapter 4 to augment each node of the Delaunay-based overlay with a limited number of carefully selected *shortcut* links. First of all, we describe three random distributions of peers in a virtual environment: uniform in a three-dimensional space, uniform on a sphere, and following a Lévy distribution [50] in three dimensional space. To our knowledge, the study of a Delaunay triangulation in the two latter contexts has never been done. We confirm through these realistic scenarios that a Delaunay-based overlay is an appealing data structure for massively shared virtual spaces. Then, we evaluate the benefits of over-

lay augmentation by simulating the underlay on top of which the Delaunay overlay is built. We show that a small number of shortcuts is sufficient to significantly decrease the delay of routing in the space.

In Chapter 5 we present a distributed algorithm for the clustering of peers in a Networked Virtual Environment (NVE) that are organized using a peer-to-peer (P2P) network based on the Delauny triangulation. The algorithm is dynamic in the sense that whenever a peer joins or leaves the NVE, the clustering will be adapted if necessary by either splitting a cluster or merging clusters. The main idea of the algorithm is to classify links between adjacent peers into short intra-cluster and long inter-cluster links.

The advantages of clustering are multiple: clustering allows to limit queries to the peers of a cluster avoiding to flood the entire network. Since clusters can be seen as a level of abstraction that reduces the amount of information/detail exposed about the NVE, clustering allows for faster navigation in the NVE and reduces the number of messages a node receives when he travels through the NVE.

CHAPTER 3

Maintaining a Delaunay-based Overlay

In this Chapter, we first propose a distributed algorithm intended to let a peer join the overlay. Second, we present an algorithm allowing peers to maintain the d-dimensional Delaunay structure in spite of ungraceful leaving or failing peers. An early version of this algorithm, for 3 dimensions only, has been published in [110]. Both sets of algorithms form a complete self-organizing self-healing Delaunay-based overlay whose usage can be extended to many other peer-to-peer applications.

3.1 Model and Definitions

An *overlay* network is a network that is built on top of another network, called the *underlay* or the physical network. Let $\mathcal{N}$ denote the set of all nodes in the underlay and $\mathcal{O} \subset \mathcal{N}$ the set of nodes in the overlay. The underlay consists of a set of nodes $\mathcal{N}$ (routers and end hosts) connected by *physical* links. The overlay consists of a set of nodes $\mathcal{O}$ (end hosts only) connected by *virtual* links.

A peer $p \in \mathcal{O}$ has two types of coordinates: (i) *overlay coordinates* p^o that indicate the position of p in the d-dimensional space of the virtual world and (ii) *network coordinates* p^u that reflect the position of the peer in the underlay. Given two peers p and q in $\mathcal{O}$, $d^o(p, q)$ denotes the *overlay distance* between p and q, which is defined as the Euclidean distance between the virtual coordinates p^o and q^o. The network coordinates can be determined at very low cost using a network coordinate system such as Vivaldi [26]. The *underlay distance* $d^u(p, q)$ is defined as the Euclidean distance between the network coordinates p^u and q^u. The underlay distance between two peers corresponds to the delay in the Internet between them. We assume that network coordinates are fixed, while the overlay coordinates can be freely chosen and changed by the user at any time. Overlay distance is a measure of the routing overhead since,

at each overlay hop, some processing must be done. Underlay distance is a measure of the delay a message will experience before reaching the destination.

If the underlay distance between two nodes p and k is less than a threshold d_t, i.e., $d^u(p,k) < d_t$, p and k are considered to be very close to each other in the underlay and are called **physical neighbors**.

The *delay penalty* can now be precisely introduced. Consider two peers p and q. Let $\Psi = \{p = h_0, h_1, \ldots, h_i = k\}$ be the set of overlay peers visited by a message sent from peer p to peer q. The sum of underlay distances between two consecutive peers in Ψ may be much longer than the underlay distance separating p and q, a case which we refer to as delay penalty:

$$\sum_{j=0}^{i-1} d^u(h_j, h_{j+1}) >> d^u(p,q)$$

In a d-dimensional overlay space, the Delaunay triangulation links peers into non-overlapping d-simplices such that the circum-hypersphere of each d-simplex contains none of the peers in its interior. Recall that 2-simplices are *triangles* and 3-simplices *tetrahedra*. Two peers p and q sharing a virtual link in $\mathcal{O}$ are *overlay neighbors*. We note $K(p)$ the set of overlay neighbors of peer p. The hypersphere which passes through all peers of a d-simplex T is noted $\mathcal{C}(T)$. The *in-hypersphere-test* verifies whether a peer is inside a hypersphere. In three dimensional spaces, a peer p is within $\mathcal{C}(a,b,c,d)$ when:

$$\begin{vmatrix} a_x & a_y & a_z & (a_x^2+a_y^2+a_z^2) & 1 \\ b_x & b_y & b_z & (b_x^2+b_y^2+b_z^2) & 1 \\ c_x & c_y & c_z & (c_x^2+c_y^2+c_z^2) & 1 \\ d_x & d_y & d_z & (d_x^2+d_y^2+d_z^2) & 1 \\ p_x & p_y & p_z & (p_x^2+p_y^2+p_z^2) & 1 \end{vmatrix} > 0$$

We assume in this thesis that peers are in *general position*, *i.e.*, no $d+1$ peers are on the same hyperplane and no $d+2$ peers are on the same hypersphere. Otherwise the triangulation would not be unique: consider 4 points on the same circle (e.g., the vertices of a rectangle). The Delaunay triangulation of this set of points is not unique. Clearly, the two possible triangulations that split the quadrangle into two triangles satisfy the Delaunay condition. Finally, we consider that the position chosen by a new peers at t is in the interior of the convex hull of $\mathcal{O}$.

3.2 Peer Insertion

We consider a new peer z joining the system at time t. We assume that z has a position in the space and knows at least one peer in $\mathcal{O}$.

In two dimensions, a known insertion technique depicted in Figure 3.1 consists of finding the triangle enclosing the new peer, then splitting this triangle into three,

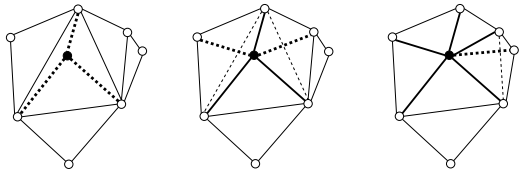


Figure 3.1: Insertion of a new peer

finally recursively checking on all adjacent triangles whether the *edge flipping* procedure should be applied [44, 46]. This is the case if the circumsphere around a triangle contains a node. In Figure 3.2, the edge flipping algorithm replaces the edge (b, c) by the edge (a, z) because $\mathcal{C}(a, b, c)$ contains the new peer z.

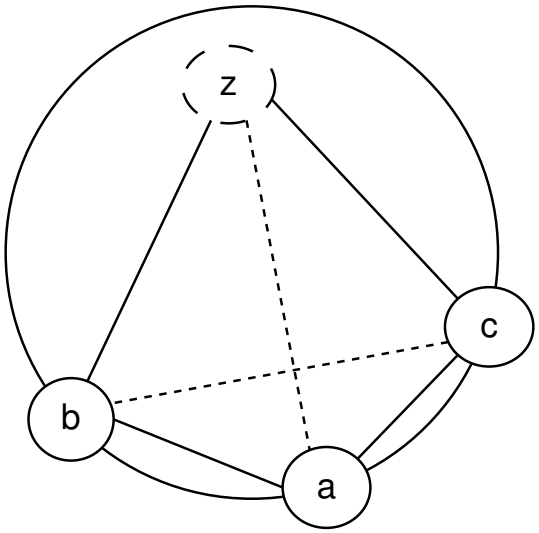


Figure 3.2: Edge flipping

Few papers study the behavior of the flipping mechanism in d-dimensional space. Most notably, a centralized incremental algorithm for the triangulation construction is proposed in [135] and, at a later time, the flipping mechanism has been proved to always succeed in constructing the triangulation [35]. We propose a distributed algorithm inspired by the edge flipping mechanism and mostly based on geometrical objects.

The algorithm works in three rounds. The first one aims to discover the enclosing d-simplex. The easiest way to achieve it consists of contacting the closest peer to the overlay position of z. Such an overlay routing is usually done in a greedy fashion. The message is forwarded to the overlay neighbor that reduces the remaining overlay distance to the destination among all neighbors. In Delaunay-based overlays, greedy routing always succeeds [11] but it requires $\mathcal{O}(n)$ time in the worst case and $\mathcal{O}(n^{1/d})$ in the average case, with n the number of nodes in $\mathcal{O}$. We tackle this issue in Section 4. The two latter rounds of the insertion process are detailed in the following.

3.2.1 Splitting the Enclosing Simplex

A peer enclosed in a d-simplex splits it into $d+1$ d-simplices. For instance, a triangle is split into three triangles as illustrated in Figure 3.1. In Figure 3.3, a new peer z belonging to a tetrahedron $T = (a, b, c, d)$ splits it into four tetrahedra $T_0 = (a, b, c, z)$, $T_1 = (a, b, d, z)$, $T_2 = (a, c, d, z)$ and $T_3 = (b, c, d, z)$.

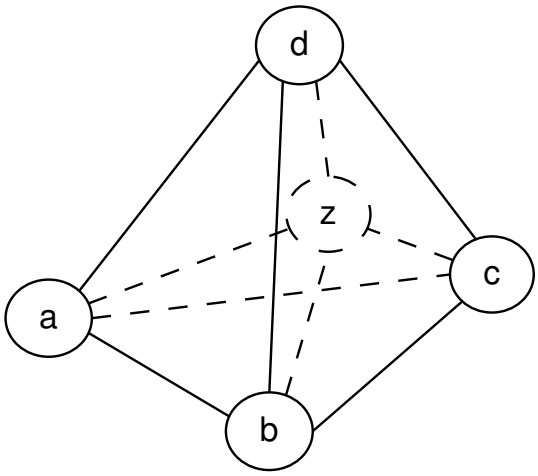


Figure 3.3: Splitting the enclosing tetrahedron

The preceding round ends when the peer z receives a description of the d-simplex T enclosing its position. This round begins by z splitting T into $d+1$ non-overlapping d-simplices. Then, it stores these simplices in a personal buffer $\mathcal{T}(z)$ in which it will further store the description of all d-simplices it is involved in. Finally, the peer z sends a `hello` message to its $d+1$ new neighbors. This message contains the former d-simplex T and the d new simplices in which the destination is involved. In the example of Figure 3.3, the peer z sends to its neighbor b a `hello` message containing T, T_0, T_1 and T_3.

3.2.2 Recursive Flipping Mechanism

We consider a peer a receiving a `hello` message from the new peer z. This message contains a d-simplex T to be discarded and d new simplices T_1, $T_2 \dots T_d$ containing both a and z.

In a first case, the simplex T does not exist in $\mathcal{T}(a)$. This unusual situation may be due to communication latencies or simultaneous events. The simplex T has been previously discarded because a peer is within $\mathcal{C}(T)$, so a invalidates this simplex[1]. The peer a should immediately inform z that T should no longer be considered as a valid d-simplex. Upon reception of this message, the new peer z should cancel its previous actions and resume the first round.

[1]Retrieving a peer that recently invalidated a former simplex can be eased by maintaining a dedicated structure called *Delaunay tree* [10].

In a normal execution, the triangulation is updated by recursively performing a flipping mechanism that splits two d-simplices into d d-simplices. This operation is called $2-d$ flip. In Figure 3.4, two tetrahedra (a, b, c, z) and (a, b, c, e) result in three tetrahedra (a, b, z, e), (a, c, z, e) and (b, c, z, e). For simplicity, we restrict the study to one simplex T_1 among the d simplices contained in the `hello` message received by peer a.

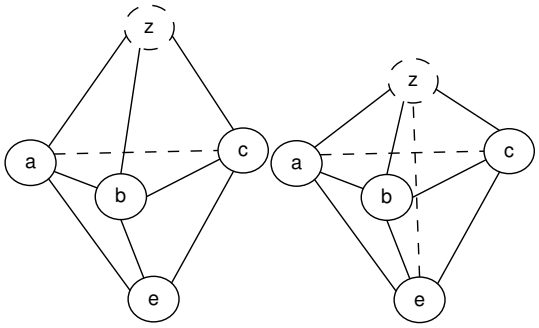


Figure 3.4: A 2-3 flip

The peer a should first determine the d-simplex $T_1' \in \mathcal{T}(a)$ such that T_1' and T_1 share one common $(d-1)$-simplex, *e.g.*, $T_1 = (a, e_0, \ldots, e_{d-2}, z)$ and $T_1' = (a, e_0, \ldots, e_{d-2}, e)$. The peer e may be considered as the opposite of z through this $(d-1)$-simplex. For instance, in two dimensions, e is the opposite of z through the edge (a, e_0), and, in three dimensions, e is the opposite of z through the face (a, e_0, e_1).

If z belongs to the circum-hypersphere of T_1', then the peer a should inform z that (1) the d-simplex T_1 should be discarded and (2) e should be considered as a new neighbor. This is achieved by a `detect` message containing both a description of e and T_1. Then, the peer a operates the $2-d$ flip, resulting in d d-simplices, noted T_{11}, T_{12}, ..., T_{1d}. One of these d-simplices does not contain a but all other should be inserted in $\mathcal{T}(a)$. Meanwhile, T_1 and T_1' are discarded.

The operation described above should be reiterated with the newly created simplices. For instance, if we consider T_{11}, the peer a should first look for the d-simplex $T_{11}' \in \mathcal{T}(a)$ sharing a $(d-1)$-simplex with T_{11}. Then, a should verify whether the new peer z belongs to $\mathcal{C}(T_{11}')$. If so, a splits T_{11} and T_{11}' into d d-simplices and sends another `detect` message to z.

This process ends until a does not split any new d-simplex anymore. The peer a should then disconnect from the peers with which it does not share any d-simplex in $\mathcal{T}(a)$. In this algorithm, all peers involved in a invalid simplex T should perform the edge flipping mechanism and send a `detect` message to z.

Algorithm 1 shows the pseudo code of the treatment at reception of a `hello` message. The first test, related to inconsistency detection, returns the peer that invalidated T (lines 2-3). In other cases, the peer receives the notification of d new d-simplices $T_1 \ldots T_d$. It puts them on a queue Q managed by a first-in-first-out policy (lines 5-6). Then it retrieves the d-simplex T_a (line 8). We consider a function `share` which takes in argument T_a and returns a d-simplex and a peer such that the d-

Algorithm 1: `hello z` T T_1 ... T_d

```
1  if T ∉ T(a) then
2      f ← detectInside (T)
3      send "cancel T, f" to z
4  else
5      for i = 1...d do
6          Q.put (T_i)
7      while Q ≠ ∅ do
8          T_a ← Q.pop ()
9          T_b, e = share (T_a)
10         if z ∈ C(T_b) then
11             T_a1 ... T_ad ← split (T_a, T_b)
12             for i = 1...d do
13                 Q.put (T_ai)
14             remove T_b from T(a)
15             send "detect e T_a" to z
16         else
17             insert T_a in T(a)
```

simplex shares with T_a a $(d-1)$-simplex and the peer is the opposite of z through this $(d-1)$-simplex (line 9). If the *in-hypersphere-test* fails, the d-simplex T_a is stored (line 17). In the opposite case, T_a is split, and the recursive process is achieved by putting the resulting d-simplices in the queue (lines 12-13). In this case, a `detect` message is sent to the new peer z (line 15).

3.2.3 Discussion

We now analyze the computational cost of this algorithm by focusing on the number of in-hypersphere tests to perform. We note k the number of neighbors of the new peer. A neighbor, except the $d+1$ peers in the enclosing d-simplex, is detected by d peers with which it forms a d-simplex. Each detection requires one in-hypersphere-test, so $d*(k-(d+1))$ operations should be performed. Moreover, there is a failing test before to end the algorithm, so $k*d$ additional in-hypersphere tests are necessary. Therefore, the total number of in-hypersphere tests is $d*(2*k-(d+1))$.

We then show that the computation task is fairly distributed among the neighbors. The worst case is as follows: one peer p_0 is linked with all of the neighbors of z, so one peer discovers all new d-simplices. As the $d+1$ peers generating the enclosing d-simplex do not require any test, p_0 should realize $k-(d+1)$ times the in-hypersphere test and d additional in-hypersphere tests to end the algorithm. Therefore, the number of in-hypersphere tests performed by one neighbor of a new peer linked to k neighbors is, at worst, $k-1$.

In order to deal with the time complexity, we are interested in the length of the worst *causality chain* that may occur in the algorithm. By causality chain, we mean a `hello` message with a `detect` message in return, this message implying a new `hello` message, which produces a new `detect` message, and so on. The worst causality chain refers to the largest possible hop distance in the overlay *before* the insertion, between the peers of the enclosing simplex and the future neighbor. Let z be the new peer and $\{p_0, p_1, \ldots, p_k\}$ the set of its neighbors after insertion. The worst case occurs when the enclosing d-simplex is $(p_0, p_1, \ldots p_{d-2}, p_k)$. The farthest peer is the peer in the middle of the path between p_{d-2} and p_k, so the peer $p_{\frac{k-(d-2)}{2}}$. The d first peers are discovered in one round and each following detected neighbor requires one new round, so we need at worst $\frac{k-d}{2}$ rounds.

3.3 Peer Deletion

We now focus on the deletion of a peer. A peer that gracefully leaves the system can quickly compute the new triangulation without itself and inform its neighbors about the new links they have to create.

However, peers may crash with no graceful behavior. The crash of a peer $z \in \mathcal{O}$ can only be noticed by a failure detector. This crash generates a *hole* in the triangulation. A very basic idea to fill it would be to let former neighbors of z reconstruct the triangulation from their knowledge of former neighbors. However, in most cases, some new connections have to be created between peers that did not know each other before the crash of z. It would be quite costly to force each peer to discover all former neighbors of the faulty peer before to reconstruct the triangulation. Rather, we propose a simple but powerful algorithm that fills the hole from its boundary to its center.

We first describe the outlines of the algorithm, then we present it in the simplest case: when z had only $d+1$ neighbors. Finally, we detail a non-trivial example in three-dimensional space.

3.3.1 Description

Assume a peer $a \in K(z)$ detecting the crash of z (see Algorithm 2). Its first task (lines 1-2) consists of extracting from $\mathcal{T}(a)$ a set $\mathcal{T}_z(a)$ of d-simplices containing z and a set $K_z(a)$ of neighbors that are, to its knowledge, affected by the crash of z. Then, the peer a retrieves the known $(d-1)$-simplices which belong to the boundary of the hole generated by the crash of z (line 3-7). We note $\mathcal{T}_z^\star(a)$ the set of these $(d-1)$-simplices.

From the set $\mathcal{T}_z^\star(a)$, the peer a builds some *candidate* d-simplices using each pair of $(d-1)$-simplices sharing a common $(d-2)$-simplex (lines 8-11). Then, each candidate

Algorithm 2: a detects the crash of z

```
1  T_z ← {T ∈ T(a) : z ∈ T}
2  K_z ← {e ∈ T : T ∈ T_z}
3  T_z* ← ∅
4  foreach T ∈ T_z do
5      remove T from T(a)
6      T* ← T \ {z}
7      insert T* in T_z*
8  foreach (T_i, T_j) ∈ T_z*, i ≠ j do
9      if |T_1 ∩ T_2| = d − 2 then
10         new_T ← (T_1 ∪ T_2)
11         insert new_T in T(a)
12         foreach e ∈ K_z : e ∉ new_T do
13             send "candidate new_T a" to e
```

simplex is transmitted with the `candidate` message to all peers contained in $K_z(a)$ but not in the simplex (lines 12-13).

We consider now a peer $b \in K(z)$ receiving a `candidate` message from the peer a. The peer b should simply verify whether it invalidates this candidate simplex T, so its only mission is to perform an in-hypersphere test with T. If the test fails — if b is not within $\mathcal{C}(T)$ — b forwards the message to the peers in $K_z(b)$ that do not participate in the candidate d-simplex T. This way, the `candidate` message turns around the hole, and no peer within the boundary may miss this new simplex.

Algorithm 3: `candidate-fail` T b

```
1  K_z ← K_z ∪ b
2  insert b in K(a)
3  T_1, T_2 ← (T_t ∈ T_z* : T_t ⊂ T)
4  foreach T_i ∈ {T_1, T_2} do
5      new_T ← T_i ∪ {b}
6      foreach e ∈ K_z : e ∉ new_T do
7          send "candidate new_T a" to e
```

If the in-hypersphere test succeeds — $b \in \mathcal{C}(T)$ — the peer b sends a `candidate-fail` message to *all the peers involved in* T. This message contains the failed simplex and a description of b itself. Upon reception of a `candidate-fail` message (see Algorithm 3), the peer a first retrieves the pair of $(d-1)$-simplices that had been used to build the failed candidate (line 3). Then, it builds some new candidate d-simplices from these $(d-1)$-simplices and the previously unknown peer b (lines 4-5). Finally, it communicates these new candidates with a `candidate` message to its neighbors on the hole boundary that are not involved with this new candidate (lines 6-7).

3.3.2 A Trivial Example

In the following, we propose a trivial case that clearly illustrates the benefits of the algorithm. We assume that z had $d+1$ neighbors, so the resulting hole is exactly the missing d-simplex in the new triangulation.

Actually, the peer z participates in $d+1$ d-simplices. Each neighbor of z is involved in d out of these $d+1$ d-simplices. For the peer $e_i \in K(z)$, the set $\mathcal{T}_z(e_i)$ contains the following d d-simplices:

$$\begin{aligned} T_0^{e_i} &= \{e_1, \ldots, e_d, z\} \\ &\vdots \\ T_j^{e_i} &= \{e_k | 0 \leq k \leq d\} \cup \{z\} \setminus \{e_j\} \quad j \neq i \\ &\vdots \\ T_d^{e_i} &= \{e_0, \ldots, e_{d-1}, z\} \end{aligned}$$

For e_i, the hole is circumscribed by the set $\mathcal{T}_z^\star(e_i)$ of all simplices in $\mathcal{T}_z(e_i)$ without z. Some candidate d-simplices can be rebuilt from any pair of $(d-1)$-simplices in $\mathcal{T}_z^\star(e_i)$. But, for each pair, the resulting d-simplex T is identical. Actually the result is only a *candidate* simplex new_T. As all peers in $K_z(e_i)$ are involved in new_T, no `candidate` message is sent, and e_i can only wait for the potential reception of a `candidate-fail` message on new_T.

So, the peer e_i sends to the d neighbors $K(z,t) \cap K(e_i,t)$ a *candidate* message containing T.

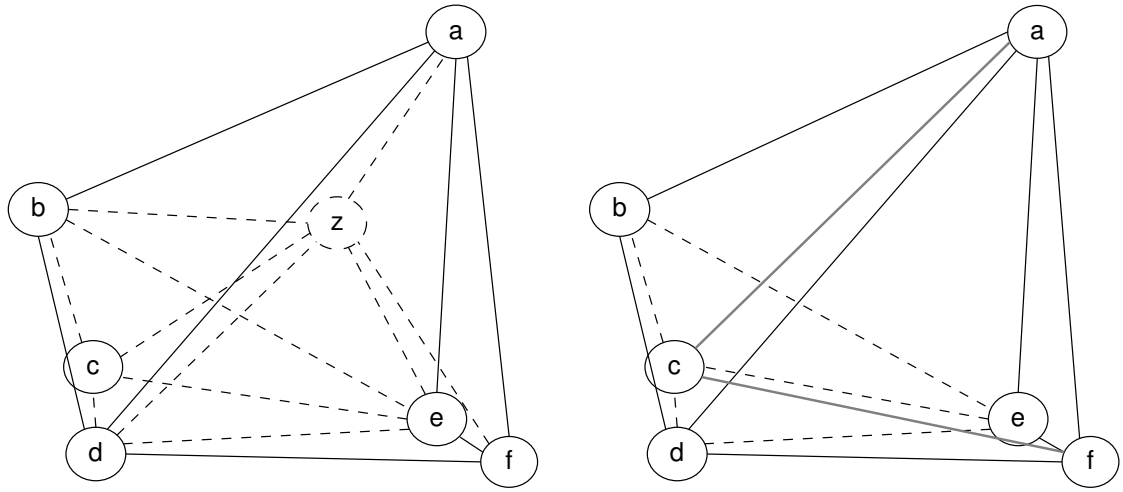
$$T = \{T_m \cup T_n\} \quad (0 \leq m, n \leq d,\ m, n \neq i)$$

For better understanding, let us rephrase this basic idea for two-dimensional space: the hole has the shape of a triangle. It is circumscribed by three edges. The three peers can be extracted from any pair of edges. So, the missing triangle can be built from any pair of edges.

In this case, no message, nor in-hypersphere test are required. The candidate simplex new_T is validated but no new connections are required.

3.3.3 A Non-Trivial Example

The following example relies on the situation depicted in Figure 3.5(a) and Figure 3.5(b). The peers a, b, d and f are in front, while the peers c and e are in the back. The faulty peer z is inside the convex hull of all these peers. The peer z is involved in 8 d-simplices (tetrahedra): (a,b,d,z), (a,b,e,z), (a,e,f,z), (a,d,f,z), (b,c,d,z), (b,c,e,z), (c,d,e,z), and (d,e,f,z). The $(d-1)$-simplices (triangles) defining the hole left by z are: (a,b,d), (a,b,e), (a,e,f), (a,d,f), (b,c,d), (b,c,e), (c,d,e), and (d,e,f).



(a) Before the deletion of z. (b) After the deletion of z.

Figure 3.5: Deletion of a node.

Consider the peer a for the rebuilding process: a is involved in 4 of the 8 tetrahedra. Consequently it only knows 4 of the 8 triangles surrounding the hole. Its candidate simplices are $T_1^a = (a, b, d, e)$, $T_2^a = (a, b, d, f)$, $T_3^a = (a, b, e, f)$, and $T_4^a = (a, d, e, f)$. They are constructed from each pair of the triangles sharing one edge.

We now focus on the candidate tetrahedron T_1^a. A `candidate` message is sent to the peer f, the only neighbor of a belonging to the boundary of the hole and not involved in T_1^a. As f is not in $\mathcal{C}(T_1^a)$ and f does not know any peers in the boundary and not in T_1^a, nothing occurs. But b acts in the same way, so the peer b sends a `candidate` message to the peer c which is known by b but not by a. As the peer c belongs to the circumsphere of the tetrahedra T_1^a, it sends a `candidate-fail` message to a, b, d and e.

The candidate T_1^a was built using the triangles (a, b, d) and (a, b, e). As they do not form a Delaunay tetrahedron together, the peer a builds two new candidate tetrahedra with the new neighbor c: $T_{11}^a = (a, b, d, c)$ and $T_{12}^a = (a, b, e, c)$. If we continue to execute the algorithm, we will see that these two tetrahedra will not be invalidated.

The final Delaunay triangulation after the deletion of z contains five tetrahedra: (a, b, c, d), (a, b, c, e), (a, c, d, f), (a, c, e, f), and (c, d, e, f). Two new connections are established: (a, c) and (f, c) as shown in Figure 3.5(b).

3.3.4 Discussion

We now analyze the computational cost of the peer deletion.

If a peer leaves the system gracefully it can quickly recompute the triangulation of its neighbors excluding itself and inform its neighbors about the new links they have to establish. The optimal computation of the new Delaunay triangulation has a com-

plexity of $\mathcal{O}(k \cdot log\ k)$ where k is the number of neighbors of the peer [30]. Other robust algorithms admit a complexity of $\mathcal{O}(k^2)$ but simpler implementations [83].

On the other hand, if a peer a crashes its neighbors have to fill the emerged hole in the triangulation without knowing all the peers on the boundary of that hole. As for the insertion of a peer we focus on the number of in-hypersphere tests to perform. In order to give an upper bound, we assume that every of the k neighbors of node a is a neighbor of every other of the k neighbors of a and that all of them notice a's crash at the same time. Each one is going to create $\binom{k}{d}$ new d-simplices and is going to communicate them to every other of the k neighbors. Every of the k nodes is going to perform $k - d$ in-hypersphere tests on every new d-simplex. Therefore, in the worst case the total number of in-hypersphere tests performed per peer is: $\binom{k}{d} * (k-1) * (k-d)$.

However, in most cases d is not much smaller than k, so that the first and third part of the term are close to one. Moreover, the k neighbors are not going to notice a's crash at the same time, therefore the `candidate` messages are not going to intersect and most of the candidate d-simplices are going to be created only once and not by every participating neighbor again.

CHAPTER 4
Underlay Shortcuts

When structured overlays were first introduced, very little attention was payed to exploit information about the underlay proximity of nodes for overlay routing. As a consequence, a message sent to a destination using overlay routing can incur a high delay penalty. To reduce the delay penalty, various proposals were made. In CAN [98], each node measures its underlay distance to a set of landmark nodes, in an effort to determine its relative position in the Internet and to construct an Internet topology-aware overlay. Pastry [100] can be enhanced by a proximity neighbor selection [21] where a node p selects among all the possible nodes with the appropriate prefix the node with the smallest underlay distance from node p. The shorter the prefix to be matched is, the more candidate nodes with the appropriate prefix exist, and the smaller is the underlay distance between the closest node and p. Hence, for any two consecutive hops in an overlay path $p_i, p_{i+1}, \ldots, p_{i+h-1}, p_{i+h}$, the overlay distance decreases, i.e., $d^o(p_l, p_{l+1}) > d^o(p_{l+1}, p_{l+2})$, while the underlay distance increases, i.e., $d^u(p_l, p_{l+1}) < d^u(p_{l+1}, p_{l+2})$. In practice, the underlay distance $d^u(p_{i+h-1}, p_{i+h})$ of the last hop dominates the total distance traveled by that message.

Another class of overlays tries to exploit the characteristics of *small world* graphs. The notion of small world, originating from social science research [80], has recently found a lot of interest in the physics, computer science and mathematics communities [136, 137]. Observations indicate that the small world phenomenon is pervasive in a wide range of settings such as social communities, biological environments, and communication networks. A small world network can be viewed as a connected graph in which two randomly chosen nodes are connected by a short path through the graph. For instance, a grid augmented by a constant number of additional long range links can exhibit small world features such as an average distance between two nodes that increases only logarithmically with the total number of nodes [60]. This property implies that one can locate information stored at any random node of a small world network by only a small number of link traversals. However, to create the additional links, global knowledge is required. Unfortunately, such approaches [55, 71, 74, 78] have some limitations. First, the physical network topology is not taken into account, and second, peers are not allowed to freely choose and

change their position in the overlay. Especially the second point is crucial if one wants to build an NVE.

In summary, we distinguish two approaches to improve overlay routing: (i) to better match the overlay and the underlay in order to reduce the delay penalty and (ii) to construct a small-world overlay with short overlay paths. However, up to now, these two approaches have not been explored together. In the following, we aim to augment the overlay with additional links in such a way that the overlay resembles a small world. Moreover, we try to favor the physical neighbors for *shortcuts* [108].

4.1 Principles

Let s be the maximal number of shortcuts and $S(p)$ the set of shortcuts of a peer $p \in \mathcal{O}$. These shortcuts are intended to cover the whole virtual world and to be physically close. Our aim is to make shortcut insertion as lightweight as possible and to optimize the coverage of the virtual world with the shorcuts. For this purpose, we use a lazy algorithm to discover new shortcuts: a peer continuously learns about the existence of other peers, (i) during the join procedure, (ii) while traveling the virtual world, (iii) or simply when forwarding a message. For each unknown peer $q \in \mathcal{O}$, the peer p retrieves its next overlay position q^o and its underlay position q^u. Then, the peer p has to decide whether peer q has to be granted as shortcut. Peer q has to fulfill two qualifications: it has to improve the virtual world coverage of $S(p)$ and it has to be physically close to peer p.

The notion of *physical neighbor* is quite simple to define if the underlay position is available. If the underlay distance between p and q is less than a threshold d_t, i.e., $d^u(p,q) < d_t$, the peers p and q are considered to be close to each other in the underlay, and so they are called physical neighbors. Note that this definition could be subject to criticisms as an isolated peer may have no physical neighbors while a very well connected one could be considered as a physical neighbor of many peers. However, as shown in Section 4.4, both situations are improbable in a worldwide application over today's Internet.

For the overlay part, it is certain that q improves the coverage of the virtual world if the number of shortcuts in $S(p)$ has not reached the maximun number allowed. Any additional shortcut naturally enhances the coverage. But if p has to discard a known shortcut to insert q, it first has to measure if this opportunity is valuable, formally if $\texttt{coverage}(S(p)) < \texttt{coverage}(S^\star(p))$ where $S^\star(p)$ contains q and $|S(p)|-1$ shortcuts out of $S(p)$ such that $S^\star(p)$ is the best configuration according to the `coverage` function.

For the `coverage` function, we propose to organize shortcuts into concentric, non-overlapping rings divided into 2^d quadrants with peer p at the center. Thus, the maximum number of shortcuts s is equal to $2^d * R$ with R the number of rings. Actually, it is likely that the peer p will not teleport to a fully randomly chosen position in the virtual world, rather the probability that it chooses a position that is relatively

close to its current position is important. Moreover, this partitioning in concentric rings approximates the probability function chosen by Kleinberg in [60].

4.2 Related Work

Several recent systems provide a self-organizing overlay for large-scale peer-to-peer applications. These systems can be viewed as providing a scalable, fault-tolerant DHT in which any item can be located within a bounded number of routing hops, using a small per-node routing table [21]. Note that peer-to-peer systems relying on DHTs cannot be used as a basis for an NVE since it is not possible for a participant to choose and to change its coordinate in the DHT and therefore in the NVE itself.

While there are algorithmic similarities between these systems, one important distinction lies in the approach they take to consider and exploit proximity in the underlying Internet. Chord [120], for instance, does not currently consider network proximity at all. Tapestry [140] and Pastry [100] exploit locality by measuring a proximity metric among pairs of nodes, and by choosing nearby nodes for inclusion in their routing tables. When choosing a node to forward a message, the remaining distance in the overlay is reduced, without moving a lot in the underlay. Only if it is unavoidable, big steps are made in the underlay. This results in first making big steps in the overlay and small steps in the underlay. Approaching the destination in this way the ratio turns; the overlay steps decrease, and the steps in underlay increase. The average total distance traveled in the overlay is only a small and constant factor larger than the distance between source and destination in the underlying network. However, these results come at the expense of a more expensive overlay maintenance protocol, relative to Chord. In CAN [98], each node measures its network delay to a set of landmark nodes, in an effort to determine its relative position in the Internet and to construct an Internet topology-aware overlay. It remains unclear to what extent the locality properties hold in the actual Internet, with its complex, dynamic, and non-uniform topology. As a result, the cost and the effectiveness of proximity-based routing in these peer-to-peer overlays remain unclear [21]. Tapestry and Pastry have a natural correlation between the overlay topology and the underlying network distance, while CAN and Chord may incur high physical hop counts for every logical hop.

None of the approaches based on the small world effect mentioned in the introduction of this chapter [55, 71, 74, 78] takes into account the underlying network topology. Moreover, these systems define the relative location and relationships of the overlay nodes themselves. Therefore they cannot be used for an NVE, since in an NVE the end-users choose their location in the overlay.

In summary some work deals with the matching of the overlay and the underlay and some other work with the construction of a small-world overlay. However, up to know these two concepts have not been explored together.

4.3 Algorithm

In the algorithm we propose in this chapter, whenever a peer p qualifies q as a physical neighbor, q is added to $S(p)$. The corresponding *field* — a pair of integers describing the ring number and the quadrant (r, q) — is retrieved from q^o and, if any, the shortcut previously stored in this field is overwritten. That is, the shortcuts are frequently renewed, reducing the probability that a shortcut is stale. Thus, it is possible to avoid to periodically ping the shortcuts to check if they are still alive.

Initially, the set of shortcuts is empty but the join requires a greedy walk through the overlay. In this case p may ask some of the nodes it has encountered for their shortcuts. This means the peers are *learning* from each other and finding shortcuts while communicating.

We have seen that local knowledge only is sufficient to find shortcuts. This and the fact that shortcuts are physical neighbors is the major difference between the algorithm we propose and the algorithm of Kleinberg [60].

These shortcuts are used for overlay routing, especially for *teleportation* and join procedures. The peer p has two types of neighbors: Delaunay-based neighbors $K(p)$ and shortcuts $S(p)$. Peers in $K(p)$ are close in the overlay whereas peers in $S(p)$ are close in the underlay. So, the idea is to use shortcuts whenever possible and to rely on overlay neighbors only when no shortcuts are available.

Consider a message to be sent to a destination d. The following is illustrated in Algorithm 4. When the routing procedure computes the next hop, p first tries to find a shortcut $p' \in S(p)$ that minimizes the remaining distance in the overlay $d^o(p', d)$ while incurring as little delay in the underlay as possible. For this purpose, the field of d is calculated (line 1). If the field is empty, the fields (r, q) in the same quadrant of interior rings (i.e., rings closer to the center) are checked (line 3-5). While shortcuts stored in interior rings $r - 1, \ldots, 1$ do not reduce the remaining overlay distance as much as a shortcut in ring r, they will reduce the remaining distance more than any overlay neighbor $K(p)$. If no appropriate shortcut is found, the overlay neighbor in $K(p)$ closest to the destination d is selected as the next hop (line 6-10).

Using this kind of routing, a message will first use shortcuts and travel long overlay distances but short underlay distances, and, as it approaches the destination, it will use overlay neighbors and travel only small overlay distances but large underlay distances. Recall that the same behavior is achieved in Pastry when using proximity neighbor selection.

4.4 Simulation Results and Evaluation

We carry out simulations to evaluate the performance improvement due to shortcuts. We show that a small number of shortcuts is sufficient to significantly decrease the

Algorithm 4: Greedy routing in an augmented Delaunay overlay

```
1  r, q ← getField(d)
2  closest ← null
3  while closest = null and r > 1 do
4  |   closest ← S(p)[r, q]
5  |   r ← r − 1
6  if closest = null then
7  |   closest ← p
8  |   foreach i ∈ K(p) do
9  |   |   if d^o(i, d) < d^o(closest, d) then
10 |   |   |   closest ← i
11 return closest
```

number of hops and the delay of the path taken: The expected number of overlay hops and the expected delay[1] of a path are no longer $\mathcal{O}(N_o^{1/d})$ but can be reduced to $\mathcal{O}(\log(N_o))$.

4.4.1 Simulation Setup

The overlay and the underlay need to be simulated together. We will first explain how to create the underlay and the overlay and then how to assign overlay nodes to the underlay.

Underlay: GT-ITM

Gt-itm [15, 16] is a widely-used tool to create synthetic network topologies. We used it to generate a 2–tier topology that consists of interconnected domains, and nodes inside each domain. Gt-itm provides us with two-dimensional coordinates for each node and the links between them. The coordinates are used as network coordinates indicating the position of a node (in the underlay).

End users that may participate in the overlay usually have one link only. Therefore a big fraction of nodes must have only one link.

The second step of the simulation is to compute all shortest paths in this topology. The delay of a shortest path is calculated by adding up the distances d^u between the nodes on the path.

Let $p_i^u, p_{i+1}^u, \cdots, p_{i+h-1}^u, p_{i+h}^u = p_j^u$ be the shortest path from peer p_i to peer p_j in the underlay. The length of this path is the sum of the Euclidean distances of the h hops

[1] In the following we will use the terms distance and delay interchangeably.

separating the two peers:

$$\sum_{k=0}^{h-1} d^u(p_{i+k}, p_{i+k+1}).$$

For our simulations we need to choose the threshold d_t that is used to determine whether two nodes are physical neighbors. Figure 4.1 shows the histogram of the underlay distances. In our case values around 80–100 ms are reasonable for d_t, thus only nodes lying in the same physical domain can be chosen as shortcuts. Note that the threshold does not need to be adapted to the number of nodes in the underlay N.

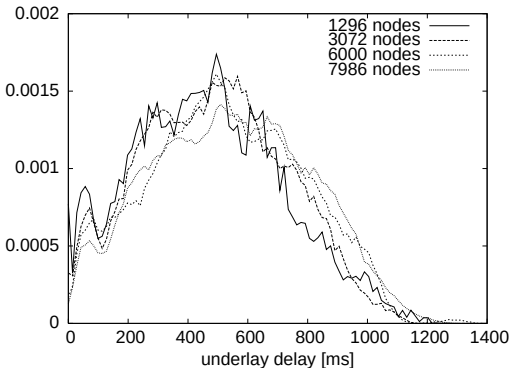


Figure 4.1: Underlay delay distribution between all pairs of nodes for topologies generated with different numbers of nodes N.

Overlay

The overlay used for the simulation uses the Delaunay Triangulation with three dimensions presented in Section 3.

The node distribution in a NVE is not uniform since nodes are organized in clusters. To obtain these clusters, we use the so-called Lévy Flight [69] that produces a random walk through space using the Lévy distribution [50] to determine the step size. The angles at the turning points are randomly chosen. The shapes of clusters generated are highly-sample dependent, meaning that if one generates clusters again and again keeping the same parameters, a great variety of different shapes is obtained. Removing the path and looking only at the (turning) points, we get a distribution of points organized in smaller and bigger clusters (Figure 4.2).

We ran simulations with uniform and clustered node distributions.

Assignment of overlay nodes to the underlay

Participants in NVEs are humans situated at the edge of the Internet. Therefore, among the nodes generated only those with a single link may participate in the

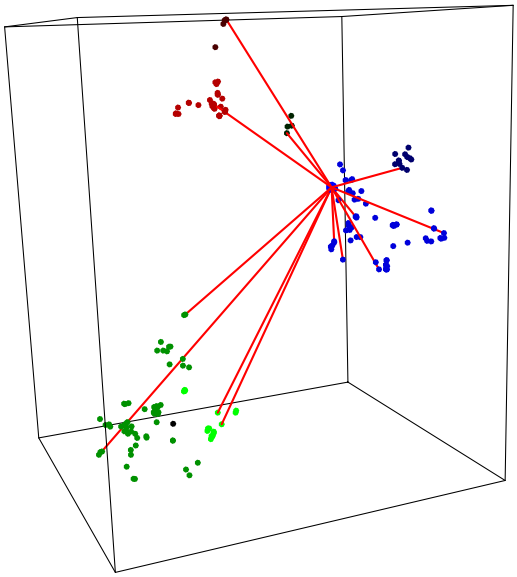


Figure 4.2: Node distribution obtained with Lévy Flight.

overlay. We simulated scenarios with different fractions of underlay nodes participating in the overlay. We could not observe that this had an influence on the results obtained.

The assignment between the nodes of the generated network and the overlay nodes is chosen at random, which will assure that peers located in the same network domain are well distributed in the overlay.

This assumption is very naive, it is very likely that there is a correlation between the location in the topology and the location in the virtual world — the overlay [13].

4.4.2 Metrics

The two most important metrics to assess the improvement due to shortcuts are the number of hops and the delay experienced by a message.

The delay is best suited to compare the performance of overlay routing, since the users care about the perceived delay. Another possibility is to measure the number of overlay hops, but this is not going to reflect whether overlay nodes are physically close or not. However, each overlay hop involves additional processing on the peers.

The memory overhead needs to be measured too, which we do by counting the average number of *shortcuts per node*:

$$S_{N_o} = \frac{\sum_{i=1}^{N_o} |S(i)|}{N_o}.$$

The value of S_{N_o} depends on d_t, since only if $d^u(p,k) < d_t$ peers p and k may consider each other as a shortcut. Increasing d_t leads to an increasing number of shortcuts per

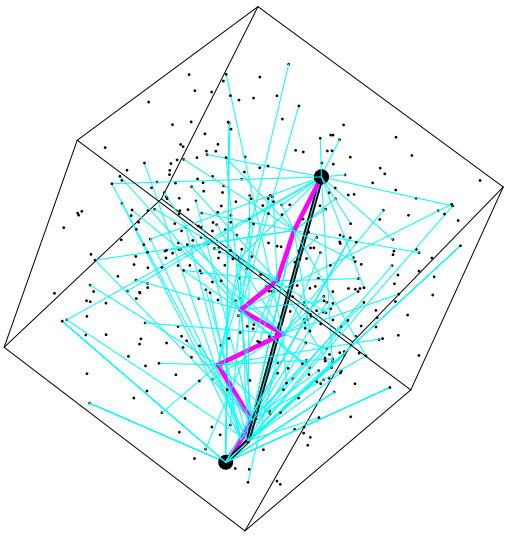
node. Only values for d_t that ensure that the shortcuts stay inside one domain in the underlay make sense, otherwise the condition that shortcuts have to be *short* in the underlay is violated.

S_{N_o} also depends on the number of rings R. If each field is occupied, $2^d R$ shortcuts are stored per node. Therefore, $S_{N_o} \leq 2^d R$.

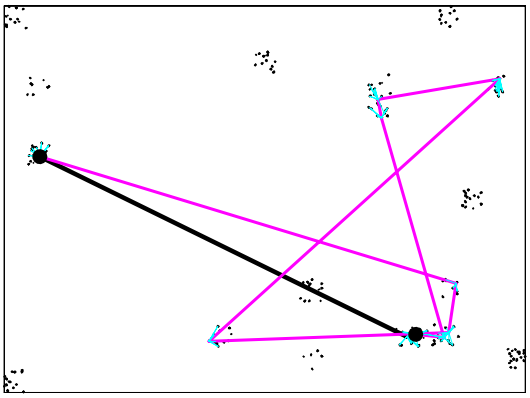
Since the peers get to know shortcuts by being part of the overlay, N_o messages are sent between random peers before the performance measures are computed. This ensures that most peers know some shortcuts.

4.4.3 Results

Let us first illustrate how our algorithm works. The example displayed in Figure 4.3 contains 1296 nodes and 3680 edges in the underlay of which 397 nodes participate in the overlay. The path displayed is the longest one, which benefits most from the shortcuts. Remember that the position in the overlay corresponds to the position of the avatar in the virtual world.



(a) Path taken in the overlay, from the point at the top to the point at the bottom. The thin lines indicate the shortcuts of the nodes visited.



(b) Paths taken in the underlay, from the point on the right to the point on the left. Grey lines: path without shortcuts; dark lines: path with shortcuts.

Figure 4.3: Path taken in the overlay and in the underlay.

Considering the *overlay* (Figure 4.3(a)), what is the difference between the paths that use the shortcuts and the ones that do not? With shortcuts, there exist overlay hops to more distant nodes, and therefore the number of hops per path is smaller. However, this does not say anything about the routes taken in the underlay and their respective delay.

Considering the *underlay* (Figure 4.3(b)), priority is given to choosing overlay hops that are nearby in the underlay. If we compare the two different paths at the underlay

level, we see that shortcuts can be very effective: The path without shortcuts uses 6 overlay hops and has a total delay of 11049 ms. Using the shortcuts, these values are reduced to 3 hops and 3418 ms. For comparison, shortest path routing in the underlay gives 2471 ms. Only the first of the three hops used is a shortcut (it is the very long one in Figure 4.3(a)).

Figure 4.4 shows the shortcuts of a node. We see that shortcuts are distributed over the entire region.

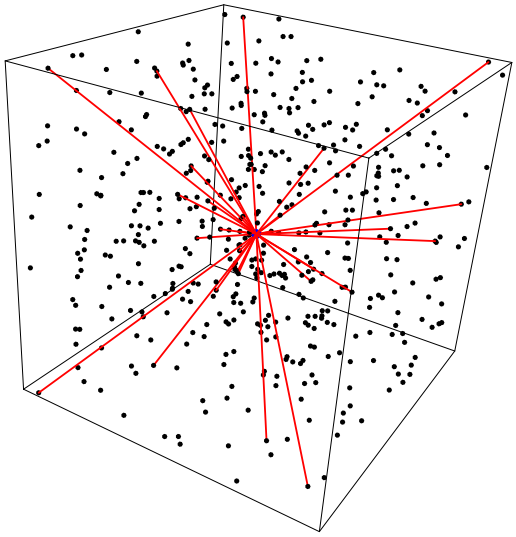
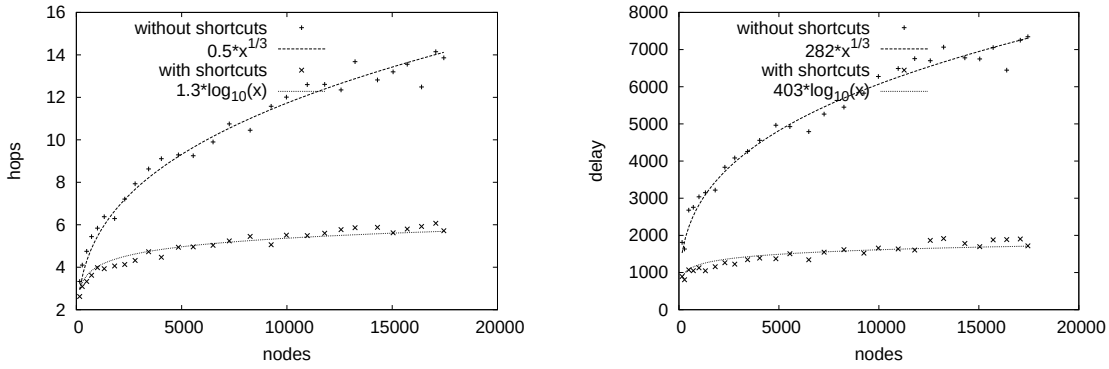


Figure 4.4: The lines show the shortcuts of the node having most shortcuts.

One goal of using shortcuts is to reduce the number of hops in the overlay. As noted earlier, the expected number of overlay hops in an overlay based on the DT is $\mathcal{O}(N_o^{1/d})$, for the three dimensional implementation used in our example it is $\mathcal{O}(N_o^{1/3})$. Using shortcuts, the expected number of hops can be reduced to $\mathcal{O}(\log(N_o))$ (Figure 4.5(a)). The same is true for the delay (Figure 4.5(b)).



(a) Average overlay hop count.

(b) Average delay.

Figure 4.5: Average hop count and delay of 100 randomly chosen paths, depending on the size N_o of the overlay network. ($d_t = 80$ ms)

To control the memory requirement introduced by the shortcuts, the threshold d_t can be adjusted. Increasing d_t leads to more shortcuts per node but also to smaller hop counts and delays. Adding only a few shortcuts already has a significant impact (Figure 4.6). After a certain point, adding more shortcuts does not significantly reduce the hop count or the delay anymore. The sweet spot lies around $d_t = 80$. It directly depends on the network topology. In the topology generated with gt-itm, the domains have a diameter of about 100 ms. Therefore it does not bring much improvement to choose a threshold bigger than $d_t = 100$. If d_t is increased so much that nodes of other domains can be considered as shortcuts, the advantage of the *short* shortcuts is lost. If d_t is too small not all nodes of the treated domain can be shortcuts, less shortcuts are chosen, and therefore the performance decreaeses.

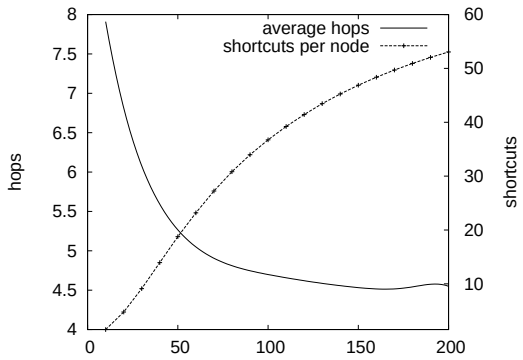


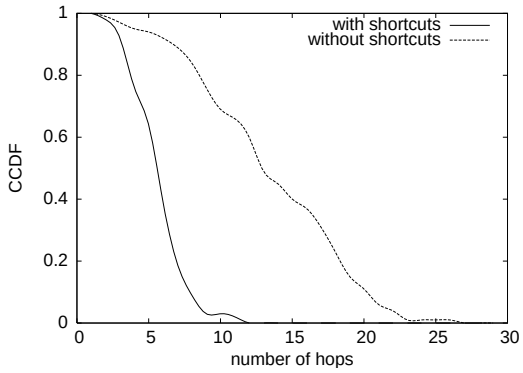
Figure 4.6: Number of shortcuts per node and the average hop count as a function of the threshold d_t ($N_o = 18000$ nodes).

Among all possible paths the very short ones, with only one or two overlay hops, will not benefit a lot (or even not at all) from the shortcuts. However, particularly the long paths with at least 10 hops benefit a lot from shortcuts. In Figure 4.7 the complementary cumulative distribution function (CCDF) of the number of hops and the delay is plotted. We see that shortcuts significantly reduce both the number of hops and the delay.

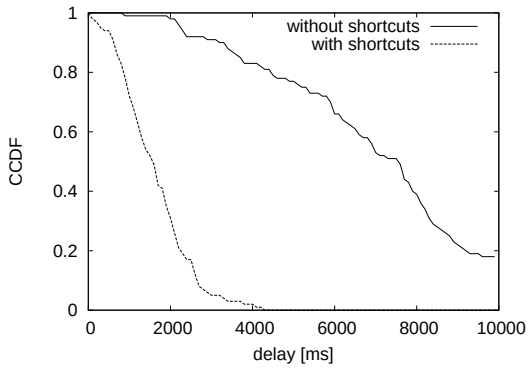
In Figure 4.7(a), for instance, we see that without shortcuts 20% of the longest paths have between 18 and 27 hops, while with shortcuts 20% of the longest paths have only between 7 and 12 hops. Looking at the distribution of a metric is more meaningful than just comparing means. In fact, it is the long paths that particularly affect the user-perceived latency. A reduction in the *tail* of the hop count and delay is therefore very important.

While not all nodes have the same number of shortcuts, the variation in the number of shortcuts is small as can be seen in Figure 4.8(a), which depicts the CCDF of the fraction of shortcut fields that are filled with shortcuts for different numbers of rings R. When we increase R, fewer fields are filled. For $R = 1$ nearly all nodes find a shortcut for each field.

The width of each ring depends on the number of rings, since the total space covered remains always the same. When doubling the number of rings R, the width of each ring is halved. Therefore, with increasing R, it will become less likely to find an ap-



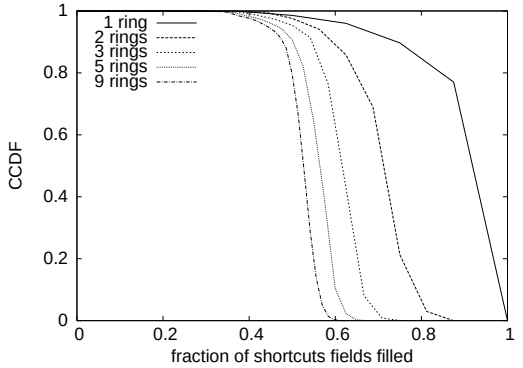
(a) The distribution of the overlay hop count.



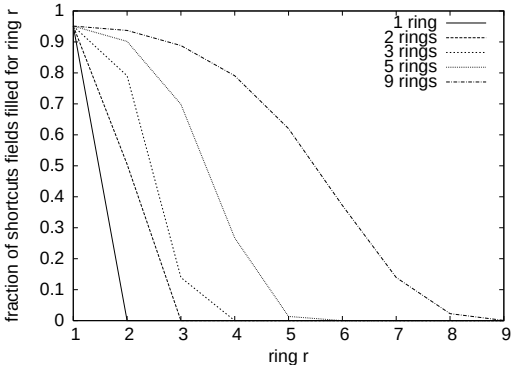
(b) The distribution of the delay.

Figure 4.7: Complementary cumulative distribution function of hop count and delay for 100 randomly chosen paths ($N_o = 18000$ nodes, $d_t = 80$ ms).

propriate shortcut for each field. This can be seen in both Figures: With increasing R, the fraction of the fields filled decreases (Figure 4.8(a)). With increasing ring number r, the fraction of the fields filled decreases (Figure 4.8(b)), which means that rings towards the center have more fields filled with shortcuts than outer rings.



(a) CCDF of the number of shortcuts per node.

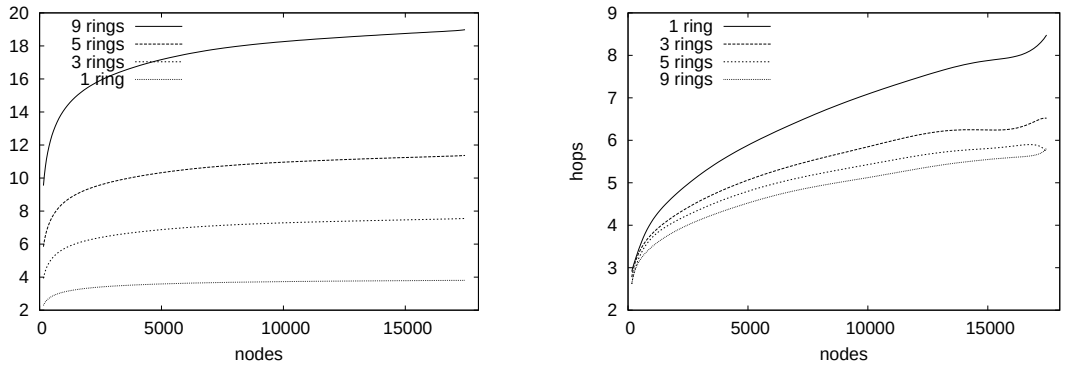


(b) The distribution of the number of shortcuts per ring r, averages over all overlay nodes.

Figure 4.8: Distribution of the number of shortcuts per node and per ring for different numbers of rings R ($N_o = 18000$ nodes, $d_t = 80$ ms).

Figure 4.9(a) shows the evolution of the number of shortcuts per node depending on the number of rings. The average number of shortcuts grows logarithmically with the number of nodes. The absolute number of shortcuts remains in the order of a few tens, which means that the storage overhead will be very low.

With increasing R, the number of hops on a path decreases, since each node stores more shortcuts. Figure 4.9(b) shows the average number of hops for 100 randomly chosen paths. The first ring introduces the biggest benefit (compared to an overlay without shortcuts); with each additional ring the added benefit decreases.



(a) The average number of shortcuts per node.

(b) The average overlay hop count.

Figure 4.9: Average number of shortcuts per node and average hop count of 100 randomly chosen paths, as function of the number of rings R ($N_o = 18000$ nodes).

By adding more rings, each peer will have more shortcuts and therefore the average hop count and delay will be reduced. To evaluate the tradeoff between hop count reduction vs. additional storage cost, we plot in Figure 4.10 the hop count reduction as a function of the number of rings, and therewith shortcuts. We see that the benefit of additional shortcuts decreases rapidly. We propose to set R to 3 or 4, which limits the shortcuts per node to 24 or 32 in the 3 dimensional case.

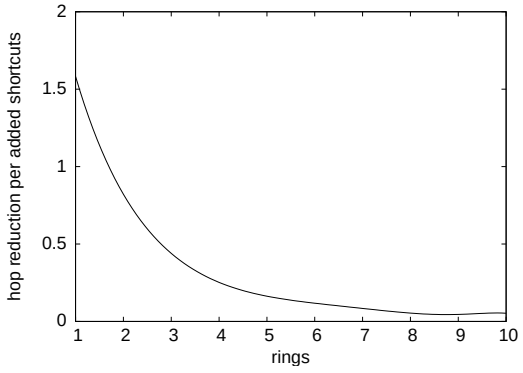


Figure 4.10: Reduction of the average overlay hops per added shortcut per node, depending on the number of rings ($N_o = 18000$ nodes).

4.5 Conclusion

As a conclusion, we have shown that already a very small number of well chosen shortcuts significantly decreases the additional routing delay introduced by the overlay network.

CHAPTER 5
Dynamic and Distributed Clustering

In this chapter we present another step of our approach to achieve scalability in NVEs that is based on the properties of a social network of people who are using the system. A social network is a set of people with interactions among them [67, 86]. In recent years, social networks have been studied with respect to properties such as degree distribution, the small world effect, the search ability, and clustering. If people interact in the virtual world as described by the social networks for the real world, they will tend to form clusters.

In this case, one can identify these clusters and limit the scope of queries to the respective cluster, avoiding to flood the entire network.

The second motivation for clustering peers is to abstract a cluster of peers into a single object and to allow faster navigation in the virtual world: not from peer to peer but from cluster to cluster. In this case, another network on top of the overlay of the nodes is needed: A *cluster overlay* network.

Another benefit using clusters is to avoid that a peer p that travels through the virtual world must receive a `hello` message from every peer that gets within its proximity, which consumes bandwidth and significantly slows down the traveling peer. Instead, it would be desirable that only one peer among all the peers of a cluster sends information about the cluster to a traveling peer p. Only if p comes so close to the cluster that it could be a part of it, p gets information about the inner structure of the cluster.

We propose a distributed and dynamic algorithm for the clustering of peers (DDC) in a virtual world based on a fully distributed P2P network [109, 111]. DDC is not only applicable to Networked Virtual Environments (NVEs), but to all proximity graphs with a clustered distribution of the nodes where each node only knows its immediate neighbors. Some possible fields of application for DDC are thus:

- Network positioning systems relying on coordinates [25, 26, 87, 90, 127]. In such systems each host is assigned a position in a d-dimensional space. The

network distance between two hosts is estimated by computing the distance using their coordinates.

- Latency-driven content delivery networks [125]. DDC can not only ease the detection of a group of nearby hosts, but also the accurate selection of the location for the replicas.
- End system multicast such as Narada [54], which is targeted towards medium-sized groups. By clustering peers and establishing multicast communication first between clusters and then within each cluster, Narada can scale to much bigger groups.
- Gossip-based (epidemic) protocols [49, 133] spread messages in a group in a randomized peer-to-peer fashion much like the spread of rumor in society, or of a contagious disease in a population. DDC could be used in such protocols to define groups of peers.

The rest of this Chapter is organized as follows: First we introduce the definitions necessary for the description of our algorithm, called DDC. Following we describe related work on clustering. In Section 5.3 we describe our algorithm DDC. In Section 5.4 we evaluate the algorithm and present some simulation results.

5.1 Principles

The set of points P is partitioned into m subsets $CL_1, \ldots, CL_m$, called clusters.

$$\bigcup_{i=1}^{m} CL_i = P$$

$$CL_i \cap CL_j = \emptyset, \quad 0 < i, j \leq n, i \neq j$$

Each peer p maintains two lists: a list $D(p)$ of its direct neighbors (e.g., its *Delaunay neighbors*) and a second list $C(p)$ of its *cluster neighbors*. Additionally, peer p stores the identifier $cid(p)$ of the cluster it belongs to. Note that $C(p) \subseteq D(p)$ and that $C(p) \subseteq CL_{cid(p)}$. $C(p) = D(p)$ if peer p is a *inner–cluster* peer, that means all neighbors of peer p are in the same cluster as peer p itself, which implies that all links incident to peer p are *intra–cluster* links.

For a peer p, let $Mean(p)$ be the mean length of connections from p to its neighbors $p_i \in D(p)$ in the Delaunay triangulation. That is

$$Mean(p) = \frac{\sum_{i=1}^{|D(p)|} d(p, p_i)}{|D(p)|}.$$

For a peer p, let Dev(p) be the standard deviation of the length of these links. That is

$$Dev(p) = \sqrt{\frac{\sum_{i=1}^{|D(p)|} (d(p, p_i) - Mean(p))^2}{|D(p)|}}.$$

$globalMean(P)$ and $globalDev(P)$ are the respective correspondents to $Mean(p)$ and $Dev(p)$, this time for all $p \in P$. They are defined as follows

$$globalMean(P) = \sum_{i=1}^{n} Mean(p_i)/n,$$

$$globalDev(P) = \sum_{i=1}^{n} Dev(p_i)/n.$$

For a peer p, let $ClusterMean(p)$ be the mean length of connections from peer p to its cluster neighbors $p_i \in C(p)$. For the definition of $ClusterMean(p)$, $ClusterDev(p)$, $globalClusterMean(P)$ and $globalClusterDev(P)$ simply $D(p)$ is replaced with $C(p)$ in the definitions of $Mean(p)$, $Dev(p)$, $globalMean(P)$ and $globalDev(P)$ respectively.

Cluster analysis has been a research topic for decades. However, most of the proposed algorithms only deal with static data, and are centralized. For our problem, we need a distributed clustering algorithm that can handle joins and leaves of peers. Also, each peer p that executes the algorithm only knows its direct neighbors $D(p)$ and not all n peers in the system.

5.2 Related Work

There are two main classes of clustering algorithms: threshold-based and density deviation-based algorithms.

5.2.1 Threshold-Based Algorithms

Kang and others [58] presented a clustering algorithm relying on the DT. The main idea is to remove Delaunay edges whose length is greater than a threshold t, and in a second step to remove clusters whose number of objects is less than a given number cn. This centralized algorithm could be adopted to a distributed one, but the main disadvantage remains: the thresholds t and cn are global values, if there exist high–density and low–density clusters, they are not recognized properly.

Based on a idea similar to the one of Kang and others, Eldershaw and Hegland [37] have also proposed a clustering criterion with the same inconvenience of the global threshold.

5.2.2 Density Deviation-Based Algorithms

Estivill–Castro et al. [39] first suggested a density-based criterion for nodes organized via a DT, referred to as *long–short* criterion. The link p_ip_j that connects two

points p_i, p_j $0 < i, j \leq n$, $i \neq j$ that are neighbors in the DT is a "short" intra–cluster link – connecting points inside a cluster – if

$$d(p_i, p_j) < Mean(p_i) - w \cdot Dev(p_i) \tag{5.1}$$

else it is a "long" inter–cluster link – connecting points in different clusters.

The idea behind this criterion is to combine spatial proximity and spatial density. In a Delaunay triangulation, a point p on the border of a cluster has a much larger value of $Dev(p)$ than an inner cluster point, since p has both, short distances to neighbors in the same cluster and long distances to neighbors that are not in the same cluster. On the other hand, peers inside a cluster have a smaller standard deviation of the distances to their neighbors since all distances inside a cluster a relatively short and of similar length.

The parameter w scales the granularity of the clusters. It does not depend on local conditions. To get satisfactory results, w must be adapted to the overall node distribution, which is not known locally. In some cases, mainly for w close to 1, the right side of the inequality (5.1) can be negative, which implies that peer p_i forms a cluster of its own.

The algorithm presented in [38] is centralized and requires global knowledge about all nodes, which allows to replace $w \cdot Dev(p_i)$ by $globalDev(P)$.

5.3 Algorithm

In this section we describe our dynamic and distributed clustering algorithm (DDC). Figure 5.1 shows the result of DDC for a set ot 1000 peers.

5.3.1 Distribution

The *long–short* criterion (5.1) must be adapted to avoid errors in the cluster determination. In the version presented in [39], two Delaunay neighbors p_i and p_j may classify their connecting link p_ip_j differently. Peer p_i may come to the conclusion that it is an intra–cluster link, while peer p_j may classify the same link p_jp_i as an inter–cluster link.

To avoid this type of error, the classification criterion must be symmetric. For this purpose, we define that a link from peer p_i to peer p_j is an intra–cluster link if

$$d(p_i, p_j) < Mean(p_i) - w \cdot Dev(p_i) \;\; and \;\; d(p_i, p_j) < Mean(p_j) - w \cdot Dev(p_j) \tag{5.2}$$

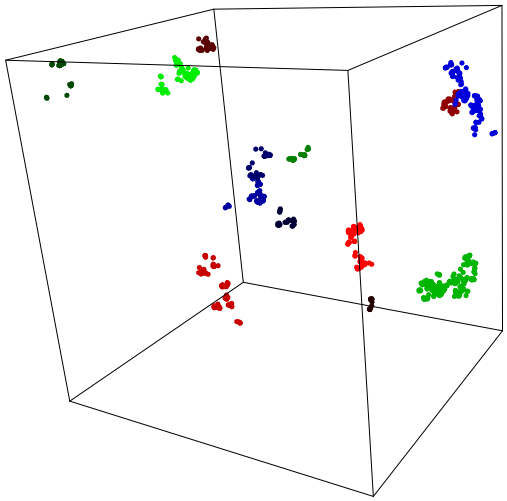


Figure 5.1: An example of 1000 peers clustered with DDC.

else the link is an inter–cluster link. This approach unfortunately leads to a highly scattered clustering because in many cases one of the two conditions is not met.

Instead, DDC uses a weighted average of the local criteria of both peers: The link from p_i to p_j is a intra–cluster link if

$$d(p_i, p_j) < \frac{Mean(p_i) - w \cdot Dev(p_i) + Mean(p_j) - w \cdot Dev(p_j)}{2} \tag{5.3}$$

else it is a inter–cluster link. Criterion (5.3) is symmetric. At the same time it classifies more links as intra–cluster links than criterion (5.2).

For the comparison of the simulation results for the three criteria see Section 5.4.

5.3.2 Dynamics

The main difference of DDC compared to [39] is that in our field of application there is no global view and that the nodes are inserted and removed dynamically, which may result in *splitting* or *merging* of existing clusters.

In the following, we describe the insertion of a new peer that will lead DDC to adapt the clustering.

The first and second peer in the NVE each form a cluster of their own. Every newly arriving peer p checks if it is near enough to its closest Delaunay neighbor c (Alg. 5, line 1) to join its cluster $cid(c)$ according to the used *long–short* criterion (line 2).

The other neighbors need not be considered in this first step. If peer p is close enough to peer c it joins the cluster of peer c (line 3+4), if not, peer p forms a cluster of its own (line 10). It is in particular not possible that peer p is near enough to anyother peer to join its cluster if it is not near enough to its closest Delaunay neighbor c. The chosen

Algorithm 5: Insertion (executed by peer p)

```
1  c ← closest(D(p))
2  if intra-cluster(c,p) then
3  |  C(p) ← (C(c) ∩ D(p)) ∪ c
4  |  cid(p) ← cid(c)
5  |  foreach pi ∈ C(p) do
6  |  |  send message to pi, to make it do:
7  |  |  C(pi) ← C(pi) ∪ p
8  |  p.redetermination()
9  else
10 |  cid(p) ← new unique cluster id
```

cid may already exist somewhere else in the network; this is not a concern as long as cid is unique in the surrounding of peer p. This can be achieved by choosing an random number out of $[1..2^{64}]$.

Due to the join of peer p, the density near p and therefore near every peer $p_i \in D(p)$, expressed by $Mean(p_i)$ and $Dev(p_i)$, changes. First, peer p updates its cluster neighbor list $C(p)$ by adding all those peers $p_i \in D(p)$ with $cid(p_i) = cid(n)$ (line 3) and notifies them so they can update their respective cluster neighbor list (line 5-7). Second, it redetermines the cluster repartition of its neighbors (line 8).

We now focus on the *redetermination* procedure (Alg. 6) which is executed by the newly inserted peer p. It therefore performs a test with the *long-short* criterion on all its Delaunay neighbors $D(p)$ (line 3+6, the function `intra-cluster` returns `true` if the link is an intra–cluster link according to the criterion used), and checks whether the result matches the neighbor classification according to $D(p)$ and $C(p)$. If it does not match, the cluster lists are updated (line 4+7), and the peers concerned continue with the *split*–procedure (Algorithm 7) or the *merge*–procedure (Algorithm 8).

Algorithm 6: Redetermination (executed by peer p)

```
1  cid ← new unique cluster id
2  foreach n ∈ D(p) do
3  |  if n ∈ C(p) and not intra-cluster(n,p) then
4  |  |  C(p) ← C(p) \ n
5  |  |  n.split(cid)
6  |  if n ∉ C(p) and intra-cluster(n,p) then
7  |  |  C(p) ← C(p) ∪ n
8  |  |  n.merge(cid(n),cid(p))
```

Split Assume that peer $n \in D(p)$ and $n \in C(p)$, where are according to the *long–short* criterion $n \notin C(p)$: in this situation peer n and peer p are too far away to be connected via an intra–cluster link (Algorithm 6, line 3). This means that peer p was

inserted so close to peer c that peer n is not any more near enough to c to be in the same cluster as peer c. In this case, it seems that the cluster containing the peers c, n and p must be split (Figure 5.2).

We still do not know if it really has to be split. The two peers n and p are connected via a long inter–cluster link, but perhaps some other peers are positioned around them in a way that keeps together the cluster. The next step tries to identify a path via intra–cluster links from peer n to peer p, if this succeeds the cluster does not need to be split.

To do so, we assume that the cluster must be split and assign a new unique cluster identifier to peer n (line 5). All peers close enough to peer n adopt this new cluster id and recursively check their neighbors in the cluster that do not already have the new cluster id.

We now consider peer $n \in C(n)$ on the reception of the *split* message from peer p (Algorithm 7).

Algorithm 7: Split(cid) (received by peer n)

```
1 if cid(n) ≠ cid then
2     cid(n) ← cid
3     foreach ni ∈ C(n) do
4         if intra-cluster (ni,n) then
5             ni. split (cid)
6         else
7             C(n) ← C(n) \ ni
```

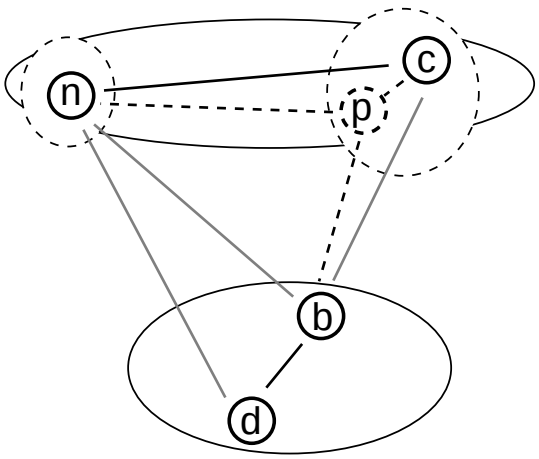


Figure 5.2: Split of the cluster cn due to the arrival of peer p. The solid lines show the links and the clusters before the split, the dashed lines afterwards.

If this is the first time peer n gets this *split* message (line 1), it updates its cluster identifier to the unique cid generated by peer p (line 2) and checks if a peer $n_i \in C(n)$ passes the *long-short* criterion (line 4). The peers that pass will all execute *split* and

change their cluster identifier to $cid(n)$. The peers that fail the test are removed from $C(n)$ (line 7) and keep their old cluster identifier $cid(c)$. This is mainly the case for the peer that executed the *redetermination* procedure (Algorithm 6) and sent the first *split* message. If the cluster really needs to be split, some of the cluster peers get the new cluster identifier $cid(n)$, the other peers keep the old one $cid(c)$. In the example presented in Figure 5.2, both peers contained in $C(n)$, peer c and peer p, fail the long-short test (line 4) and are therefore removed from $C(n)$ (line 7). The *split*–procedure terminates here.

It might happen that peer c and peer n are too far away to be connected by an intra–cluster link but nevertheless are part of the same cluster because there exists a path of intra–cluster links connecting them. In this case the recursive *split* message finds its way back to peer n, and all peers of the cluster get the new cluster number $cid(n)$. The cluster keeps its form and is not split but simply renamed.

Merge Assume that peer $n \in D(p)$ and $n \notin C(p)$ but according to the *long–short* criterion $n \in C(p)$: in this situation peer n and peer p are close enough to be connected via an intra–cluster link, but they are not part of the same cluster yet (Algorithm 6, line 6). That means peer p was inserted in between two existing clusters $CL_{cid(n)}$ and $CL_{cid(c)}$ and interconnects them (Figure 5.3).

We now consider peer $n \in C(p)$ after the reception of the *merge* message from peer p (Algorithm 8). If this is the first time peer n gets this *merge* message (line 1), it changes its cluster identifier $cid(n)$ to $cid(c)$ (line 2) and tells every peer $n_i \in C(n)$ to propagate the new cluster identifier $cid(n) = cid(c)$ to their respective cluster neighbors $C(n_i)$ (line 4+5). The sender of the *merge* message, peer ni, is added to $C(n)$ (line 8). This results in a merge of cluster $cid(n)$ and $cid(c)$ to cluster $cid(c)$, all peers of the cluster $cid(n)$ change their cluster identifier to $cid(c)$.

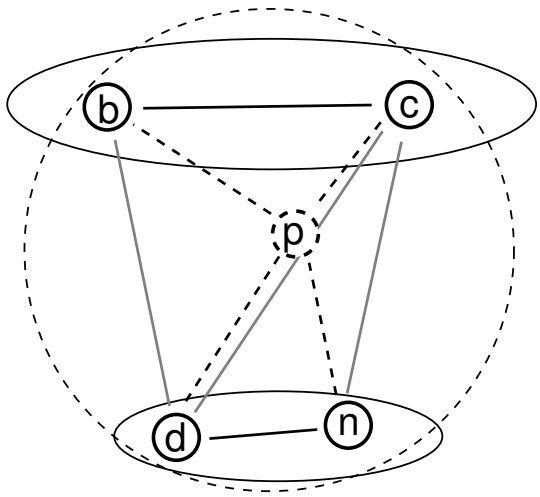


Figure 5.3: Merge of the cluster $CL_{cid(c)}$ and $CL_{cid(n)}$ due to the arrival of peer p. The solid lines show the links and the clusters before the split, the dashed lines afterwards.

Algorithm 8: Merge($oldcid$,$newcid$) (received by peer n)

```
1 if cid(n) = oldcid then
2     cid(n) ← newcid
3     foreach ni ∈ D(n) do
4         if ni ∈ C(n) then
5             ni. merge (oldcid,newcid)
6         else
7             if intra-cluster (n,ni) then
8                 C(n) ← C(n) ∪ ni
```

The cluster redetermination in case of a peer departure functions in a similar way as the case of a peer joining: Either the cluster of the closest peer – who handles the departure in the Delaunay triangulation – must be split or, it must be merged with another cluster.

5.3.3 Proof of termination

Each peer p that joins has one closest neighbor c only. Either peer p does not join c's cluster, in this case the algorithm ends, or peer p joins c's cluster and launches the *redetermination* procedure (Algorithm 6), which is only executed once per join or leave event.

At most $|C(p)| - 1$ *split* messages are sent by peer p to all cluster neighbors except to node c.

Recursively, a receiver n of a *split* message sends at most $|C(n)| - 1$ new *split* messages. Out of these t messages ($t = |D(p) \cap D(n)|$) are directly ignored. Eventually, every peer of the cluster got one message, the algorithm terminates here since the condition (Algorithm 7, line 1) is false. This is a simple "flooding" of the cluster concerned, whereas no peer runs the *split* procedure twice and sends messages a second time.

Exactly the same statements are true for the *merge* procedure. The cluster concerned is simply renamed, while it is assured that no peer executes *merge* twice (Alg. 8, line 1).

5.4 Simulation Results and Evaluation

In this section, we describe the simulation environment, the choice of the clustering criterion and the choice of the clustering threshold w. Moreover, we show that DDC deals well with all sorts of node distributions.

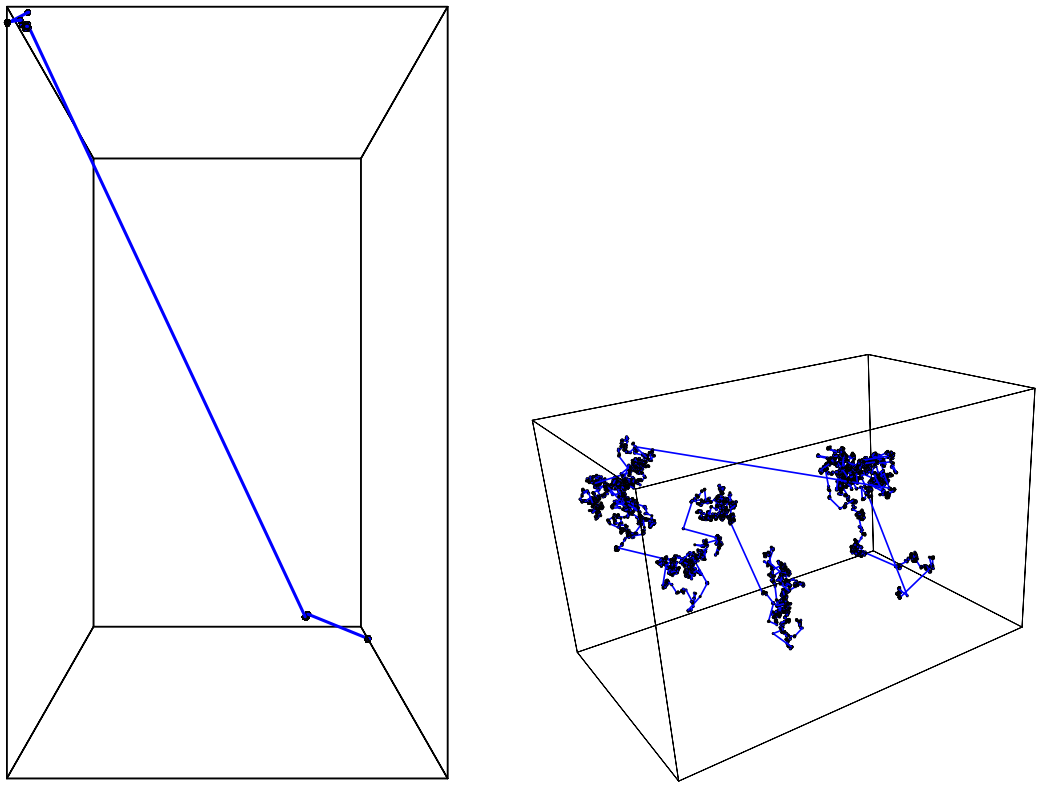
5.4.1 Generation of a Clustered Peer Distribution

To evaluate DDC, first clusters of peers must be generated. For this purpose, we use the so called Lévy Flight [69] that produces a random walk through the plane or the space where the Lévy distribution [50] determines the step size.

The Lévy distribution has mathematical properties that discourage a physical approach. Generally it may be expressed as

$$L(x) = \frac{1}{\pi} \int_0^\infty e^{-\gamma q^\beta} \cos qx \quad dq$$

and is known as *symmetrical Lévy stable distribution* of index β $(0 < \beta \leq 2)$ and scale factor γ $(\gamma > 0)$. For simplicity we set $\gamma = 1$. It has infinite variance and an analytical form known only for a few special cases, as the Cauchy and the Gauss normal distribution. Its tail has a power-law decay, it is heavy tailed.



(a) With $\beta = 0.55$ one big step dominates the distribution.

(b) With $\beta = 1.95$ the clusters are near together but still sharply separated.

Figure 5.4: The results of two Lévy Flights with 10,000 nodes each. The lines show the way of the flight.

Note that the characteristic size of the system is the size of the largest step and that the flight is self-similar at higher magnifications. The shapes of clusters generated are highly sample-dependent, meaning that if one generates clusters again and again keeping the same parameters, a great variety of different shapes is obtained. Removing the path and looking only at the turning points, we get a distribution of points

having smaller and bigger clusters. The degree of clustering depends on β, decreasing it means that long jumps are made longer, while short jumps are made shorter. The tail of the distribution gets heavier. Yet the flight is rescaled to fit in a graph of a fixed size, so that the visible effect of increasing β is that the clusters become identifiable. Each cluster is made to appear increasingly tightly packed and therefore increasingly separate from neighboring clusters [75].

The implementation works by inverting the distributive function approximating the Lévy distribution, which is given by

$$L_s(\xi) = \xi^{-\frac{1}{\beta}} - 1,$$

where ξ is uniformly distributed over $[0, 1]$ [50].

The angles at the turning points are chosen randomly: Set up a coordinate system (z, ϕ) where z is an arbitrary axis, ($z = -r, \ldots, r$, where r is the sphere's radius), and where ϕ is the longitude, which runs between 0 and 2π. To generate a random point on the sphere, it is necessary only to generate two random numbers, z and ϕ, each with a uniform distribution. To find the latitude θ of this point, note that $z = r\sin(\theta)$, so $\theta = \arcsin(\frac{z}{r})$; its longitude is simply ϕ. In rectilinear coordinates, $x = r\cos(\theta)\cos(\phi)$, $y = r\cos(\theta)\sin(\phi)$, $z = r\sin(\theta) = z$. (x, y, z) are not independent but constrained by $x^2 + y^2 + z^2 = r^2$.

Figure 5.4 shows the results of two Lévy Flights with $\beta = 0.55$ and $\beta = 1.95$ containing 10,000 points each. In both cases the clusters are sharply separated. For $\beta = 0.55$ due to the "very" heavy tail of the distribution one big step dominates, whereas for $\beta = 1.95$ the clusters are nearer together but still well defined. To simulate a NVE, values for β near 2 seem to be appropriate.

5.4.2 The choice of the clustering criterion

In Section 5.3 we introduced three different clustering criteria. The metric to compare the clustering results obtained with DDC using the three criteria is defined as follows: Remember the definition of $globalMean(P)$, $globalDev(P)$, $globalClusterMean(P)$ and $globalClusterDev(P)$ (refer to Section 4.1). If the clustering is well done $globalClusterMean(P)$ is lower than $globalMean(P)$ since the long links from nodes lying on the clusters boundaries to non–cluster neighbors are cut off. The same is true for $globalClusterDev(P)$ and $globalDev(P)$. However one can not postulate that the better the clustering is done, the bigger the quotients $\frac{globalMean(P)}{globalClusterMean(P)}$ and $\frac{globalDev(P)}{globalClusterDev(P)}$ get, since in this case the partition of one node per cluster ($m = n$) would be optimal. Therefore the total number of clusters m has to be considered as well.

The metric to compare the different criteria is defined as follows:

$$ClusteringQuality = \frac{\frac{globalMean(P)}{globalClusterMean(P)}}{m}.$$

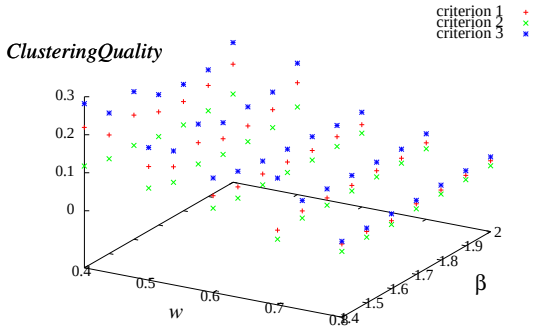


Figure 5.5: $ClusteringQuality$ dependent on w and β for the three clustering criteria (average values of 30 runs with 1,000 nodes each).

In Figure 5.5 the values of $ClusteringQuality$ are plotted. The values are independent of β and decrease with decreasing w. This is as expected, since β has no influence on the cluster recognition. Note that the quotient is independent of β because both the $globalMean(P)$ and the $globalClusterMean(P)$ are in the same way dependent on β. With decreasing β these two values increase. w stands for the cluster recognition threshold, for its optimal value refer to the next section.

As expected in Section 5.3 the criterion (5.3) is the best suited one. In the following Sections we therefore run the simulations with this criterion only.

5.4.3 The choice of the clustering threshold w

Since no global view of the NVE exists, the optimal value for w cannot be derived from the peer distribution but must be chosen in advance.

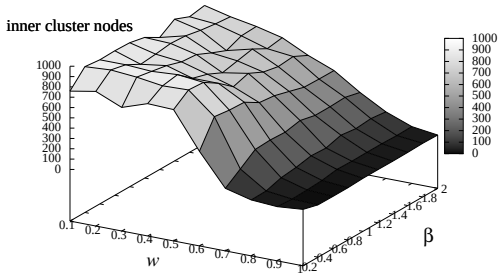
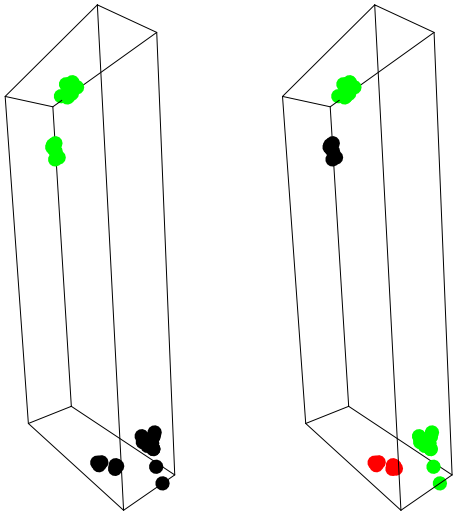


Figure 5.6: The number of inner–cluster nodes as a function of w and β (average values of 10 runs with 1,000 nodes each).

Values between 0 and 1 are reasonable for w. To find a value for the distributions created with a Lévy Flight, we run simulations for $w = 0.1 \ldots 1.0$ with step size 0.1 and $\beta = 0.2 \ldots 2.0$ with step size 0.05. As can be seen in Figure 5.6 the results are independent of β. Thus, DDC is well adapted to all possible degrees of clustered node distribution. The results depend on w, in the Figure a "knee" at $w = 0.6$ is observable. The number of inner–cluster nodes (all nodes n for which $C(n) = D(n)$) is stable for $0 < w < 0.6$ and begins to decrease at $w \approx 0.6$. For $w < 0.6$ there exists

only one cluster in most cases, therefore the number of inner–cluster nodes is equal to the total number of nodes. The same observations as for the number of inner–cluster nodes can be made for the number of clusters and the number of cluster neighbors per node. Therefore, we come to the conclusion that the choice $w = 0.6$ is suitable for determining clusters in distributions generated with the Lévy Flight.

Figure 5.7 shows an example with only 50 peers to point out the influence of w on the clustering result. In this case both choices for the threshold w make sense, depending on one's needs.



(a) $w = 0$ (b) $w = 0.6$

Figure 5.7: Clustering results with $w = 0$ (two clusters) and with $w = 0.6$ (four clusters).

5.4.4 Difficult cases

In section 5.4.2 we already showed that DDC deals well with all sorts of node distributions generated with the Lévy Flight. Moreover, we simulated two special cases, first two clusters that are connected via a bridge of nodes, and second a high–density cluster surrounded by some sparse points.

Bridges between clusters If two clusters are connected via a bridge (Figure 5.8) the two clusters can not be distinguished using local knowledge. With the help of the clustering criterion it is possible to assume that long links are links between peers in different clusters, because they are too far away to be in the same cluster. Whereas we assume that short links are always intra–cluster links, which is not always true. Sometimes these very short links can form *bridges* between clusters. These bridges are not recognized properly since only local (one hop) knowledge is available: clusters connected by bridges are seen as one cluster by the *merge* procedure. For the

application in a NVE that does not matter because we assume that peers acting in the NVE do not deliberately form bridges which does not give any benefit to them.

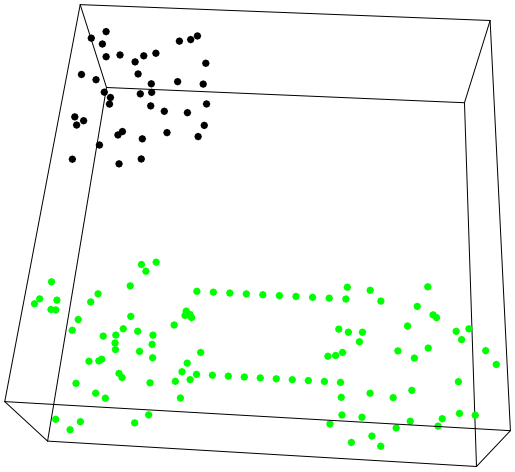


Figure 5.8: Two clusters connected by two bridges are not properly distinguished.

Density variations In Figure 5.9 one cluster lies inside some relative sparse points. The algorithm detects the inner high–density cluster and also clusters the sparse points, totally independent of their shape. That is impossible to an algorithm with a global threshold.

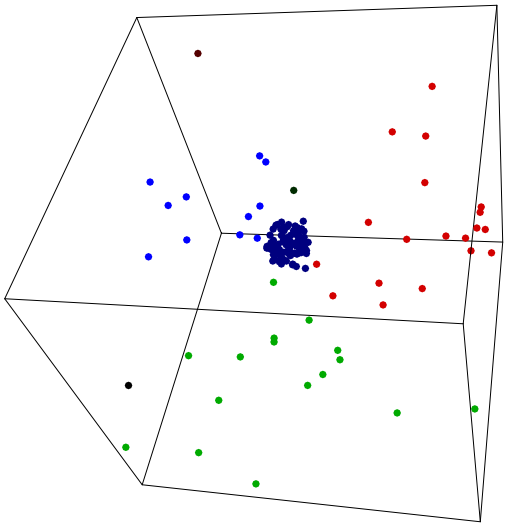


Figure 5.9: One high–density cluster containing 100 points, and some 50 sparse points around it.

5.5 Conclusion

In this Chapter we have introduced algorithms for the dynamic and distributed clustering of peers. Essentially for these algorithms is the symmetric clustering criterion that allows two peers to classify their common linnk it in the same way.

CHAPTER 6
Conclusion of Part I

"That theory is worthless. It isn't even wrong!"
– Wolfgang Pauli –

In this part of the thesis we presented a set of algorithms for the dynamic maintenance of a distributed overlay network based on the Delaunay triangulation of the entities. The entities can have a position in any $d-$dimensional space. We consider here the arrival of new entities and the ungraceful leave of an entity. We show that the algorithms we propose succeed in re-constructing the Delaunay triangulation through a self-stabilization scheme. In two dimensions it is possible to rely on an angular feature that is not available in higher dimensions. However, considering three to d dimensions the algorithms remains unchanged.

We showed how a Delaunay-based overlay can be augmented in very simple way by additional shortcuts to build a small-world overlay that will reduce both, the average number of hops and the delay. When choosing shortcuts, we exploit the fact that nodes close to each other in the underlay may be far away in the overlay.

The idea of shortcuts is not limited to Delaunay-based overlays but can be used in other structured overlays. A promising candidate is CAN, where each node only knows overlay nodes that are close-by, and shortcuts can be used to increase the overlay distance traveled at each hop.

We moreover presented DDC, a dynamic and distributed algorithm to cluster nodes in a peer-to-peer–based virtual world. Clusters of different shapes and density are detected. The dynamic approach copes with peer insertion and deletion without having to restart the cluster detection from scratch. It tests locally if the density changed so much that a cluster has to be split or that two cluster have to be merged. It is very robust and can be adapted to all sorts of distributions with one tuning parameter.

Concerning the clustering one open issue remains: some thresholds are fixed constants and need to be fixed dependent on the properties of the system.

One motivation for the clustering is to define a natural limit for the scope of a query in order to avoid flooding the entire network of peers. The other motivation is to procure an abstract view of a group of peers – a cluster – to other peers that are far away from this cluster.

Part II

Measurements of real world peer-to-peer networks

CHAPTER 7
Introduction

"Definition of Statistics: The science of producing unreliable facts from reliable figures."
– Evan Esar –

Peer-to-Peer Systems have seen a tremendous growth in the last few years, and peer-to-peer traffic makes a major fraction of the total traffic seen in the Internet. The dominating application for peer-to-peer is file sharing. Some of the most popular peer-to-peer systems for file sharing have been Napster, FastTrack, BitTorrent, and eDonkey, each one counting a million or more users.

An appealing solution to organize peers and content in a overlay network are the so called Distributed Hash Tables (DHTs). They have been actively studied in the literature and many different proposals have been made on how to organize peers in a DHT. However, very few DHTs have been implemented in real systems and deployed on a large scale. One exception is KAD, a DHT based on Kademlia, which is part of the eDonkey network, a peer-to-peer file sharing system with several million simultaneous users.

Since all these systems are mainly used by home-users and since the content shared is typically copyright-protected, the users of these systems often stay only connected as long as it takes for them to download the content they are interested in. As a result, the user population of these peer-to-peer systems is highly dynamic, with peers joining and leaving all the time. This major issue in peer-to-peer networks is called *churn*.

This thesis presents an in-depth study of KAD. We want to understand KAD with the goal to suggest some improvements of the present algorithms. The focus of our interest lies in the user behavior, the duration of their online sessions, the frequency with which they reconnect to the network, their availability, and the overall time span they use KAD. These metrics express to some extend the satisfaction of the users with the existing system, but moreover they can be used to tune the parameters chosen by the developers of KAD and thus improve the performance or reduce the overhead.

In order to improve the performance of the publish and search algorithms in KAD we want to analyze in detail the design of those algorithms, and especially the implementation of the content lookup procedure of KAD in the aMule client. These analyses are going to be complemented by measurement results of the publish and search overhead. We do this again with the goal to improve the performance of KAD, indeed we present a new content lookup scheme that can be integrated in aMule without fundamental modifications of the client.

While measuring the user behavior of KAD we stumbled upon a set of vulnerabilities that made the measurements possible in the first place. Neither is the privacy and anonymity of the users guaranteed, nor is the data safe from being corrupted or polluted by other parties.

7.1 Contributions

To learn about the users' behavior and especially about churn we developed **Blizzard**, a peer crawler for KAD. The speed of *Blizzard* allows us to crawl the entire KAD ID space, which was never done before. We have been crawling a subset of KAD continuously for almost six months at a high frequency, and we obtained information about geographical distribution of peers, session times, peer availability, and peer lifetime. We found that session times are Weibull distributed and show how this information can be exploited to make the publishing mechanism much more efficient.

Blizzard allows us to measure the user behavior but not the shared content in KAD. Therefore we designed and implemented **Mistral**, a content spy that can capture up to ten million references to published content in several hours. In a first evaluation, we noticed that publishing new content in a KAD system is much more expensive than searching and retrieving existing content. Indeed, measurements show that of all the Internet traffic generated by KAD-based peer-to-peer networks, 90% is for publishing and 10% for retrieving existing files. Moreover, the most frequently published keywords are meaningless stopwords. We propose to add a stopword filtering mechanism to the search and publish procedures of KAD-based peer-to-peer systems.

While implementing Mistral, we learned a lot about KAD, we have been both fascinated and frightened by the possibilities KAD offers. It is indeed simple to **exploit** KAD, e.g., mounting a Sybil attack in KAD is not prevented by any mechanisms and allows to compromise the privacy of KAD users, to compromise the correct operation of the key lookup, to "delete" content, and to execute a DDoS attack with very little resources. Also, overwriting content with a so called *pollution attack* is easily feasible. We tested this type of attack against the Storm worm botnet that uses a KAD protocol for communication. It is one of the first botnets that relies on a peer-to-peer protocol for communication. Botnets, networks of compromised machines under a common control infrastructure, are commonly controlled by an attacker with the help of a central server: all compromised machines connect to the central server and wait for commands.

Among the different **publishing schemes**, KAD adopts a publishing node centric approach: the responsibility of the content and its maintenance is on the publishing node, while the references to it are announced and stored in the peer-to-peer system. In order to make the references available independently of node dynamics, a peer in KAD publishes multiple copies of a reference by selecting different nodes around the target that is determined by the key of the reference. As time goes by, some replicas may disappear, or new peers may arrive and take place in between the copies.

The actual presence of the references is then scattered: some entries in the routing tables may be missing since the peers arrived recently, or may be stale since the peers already left the system. In case of content retrieval, where these references are searched around the target where they should be published, robust search mechanisms are necessary. We developed a qualitative analysis of the current implementation to understand the impact of the design parameters on the latency of the overall content publishing/retrieval process. We obtained through measurements many interesting properties of the KAD system, such as the probability that an entry in the routing table is stale, or the round trip delay of the messages: Finally we evaluate through measurements the key performance metrics, such as overall content retrieval latency, the number of hops needed, and message overhead of the content retrieval process.

Network Coordinates allow to estimate the latency among a large number of hosts in a scalable way. Recently, Azureus, a popular implementation of BitTorrent, has implemented network coordinates. We have developed a crawler that allows us to obtain from over one hundred thousand peers running Azureus their network coordinates and to measure the network and application level round trip times to these peers.

Our measurements confirm that network coordinates allow to correctly estimate the round trip time between two peers. Our measurement also show that the round trip times from our crawling host to a set of peers located in the same country can vary between a few tens of milliseconds to more than one second. This high variance is due to the large buffers in the ADSL access links, which can increase the round trip time by hundreds of milliseconds. As a consequence, network coordinates and round trip estimations in general cannot be used to select peers that are “nearby”, such as peers connected to the same ISP or located in the same country.

CHAPTER 8
Background and Methodology

KAD is a Kademlia-based [77] peer-to-peer DHT routing protocol that is implemented by several peer-to-peer applications such as Overnet [89], eMule [34], aMule [1], AZUREUS [6], and lately the Storm Worm [51]. The two open–source projects eMule and aMule do have the largest number of simultaneously connected users since these clients connect to the eDonkey network, which is a very popular peer-to-peer system for file sharing. Recent versions of these two clients implement the KAD protocol.

Similar to other DHTs like Chord [120], Can [98], or Pastry [100], each KAD node has a global identifier, referred to as a KAD ID, which is a 128 bit long random number generated using a cryptographic hash function.

Peers are identified by the so-called KAD ID, which was up to now assumed to be persistent. However, we observed a large number of peers that change their KAD ID sometimes as frequently as after each session. This change of KAD IDs makes it difficult to characterize *end-users*. However, by tracking end-users with static IP addresses, we could measure the rate of change of KAD IDs per end-user and the end-user lifetime.

However, in most cases, using the KAD ID, a particular peer can be tracked even after a change of its IP address. This is important since many ISPs reassign IP addresses to their customers as often as once a day.

8.1 Routing in Kademlia

Routing in KAD is based on prefix matching: Node a forwards a query destined to a node b to the node in his routing table that has the smallest XOR-distance to b. The XOR-distance $d(a, b)$ between nodes a and b is $d(a, b) = a \oplus b$. It is calculated bitwise on the KAD IDs of the two nodes, e.g., the distance between $a = 1011$ and $b = 0111$ is $d(a, b) = 1011 \oplus 0111 = 1100$, and the distance between $a = 1011$ and $c = 1100$ is

0111. Thus a is closer to c than to b since $d(a,b) = 1100 > d(a,c) = 0111$. The fact that this distance metric is symmetric is an advantage compared to other systems, e.g. Chord, since in KAD if a is close to b, then b is also close to a. Therefore, a node a that announces its existence to a node b might be added by node b to its routing table.

The entries in the routing tables are called *contacts* and are organized as an unbalanced *routing tree*: A peer P stores only a few contacts to peers that are far away in the KAD ID space and increasingly more contacts to peers closer in the KAD ID space. For further details of the implementation see [77, 122]. For a given distance, P knows not only one peer but a *bucket* of peers. Each bucket can contain up to ten contacts, in order to cope with peer churn without the need to periodically check if the contacts are still online. Each contact consists of the node's KAD ID, IP address, TCP and UDP port. The *left* side of the routing tree contains contacts that have no common prefix with the node a that owns the routing tree (XOR on the first bit returns 1). The *right* side of the routing tree contains contacts that have at least one prefix bit in common. This tree is highly unbalanced The right side of each tree node is (recursively) further divided into two parts, containing on the left side the contacts having no further prefix bit in common, and on the right side the contacts having at least one more prefix bit in common. A *bucket* of contacts is attached to each leaf of the routing tree. Each bucket can contain up to ten contacts in order to cope with peer churn without the need to periodically check if the contacts are still online.

To route a message towards its destination the next hop is chosen from the bucket with the longest common prefix to the target. Routing to a specific KAD ID is done in an iterative way, which means that each peer, on the way to the destination, returns the next hop to the sending node. While iterative routing experiences a slightly higher delay than recursive routing, it offers increased robustness against message loss, and it greatly simplifies the crawling of the KAD network.

8.2 Two-Level Publishing in Kademlia

A **key** in a peer-to-peer system is an identifier used to retrieve information. KAD distinguishes between two different keys:

- A **source key** that identifies the content of a file. It is computed by hashing the *content* of a file.
- A **keyword key** that classifies the content of a file and is computed by hashing a single token from the *name* of a file.

In KAD each key is not published just on a single peer that is numerically closest to that key, but on 10 different peers whose KAD ID agrees at least in the first 8-bits with the key. This zone around a key is called the *tolerance zone*.

Figure 8.1 shows an example of the publishing process. A peer wants to publish a file with the name `the_matrix`. This filename will result in two keywords, "the"

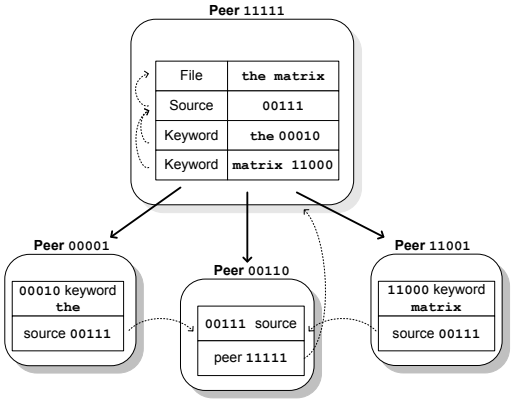


Figure 8.1: Sketch of the 2-level publishing scheme

and "matrix". All relevant references to the original file are generated, such as the source key and the the keywords with the attached metadata. Next, the keywords "the" and "matrix" are published, pointing to the source key. Finally, the source key is published, with a pointer to the publishing peer.

Keys are periodically republished: **source keys** every 5 hours and, **keyword keys** every 24 hours. Analogously, a peer on which a source key or keyword key was published will delete the information after 5 and 24 hours respectively. Re-publishing is done in exactly the same way as publishing.

The KAD system is designed to prevent free-riding: anyone who retrieves a file from KAD also becomes a server for that file, and he publishes this fact to the rest of the world. Thus, new publications are a consequence of successful retrievals.

The peer that accepts the publish message for a keyword returns the load factor to the publishing peer. The load factor takes values between 0 and 100 and is computed as a function of the number of publications for the specific keyword and the total number of publications the peer received and stored. If the load factor is below 20, the default republishing delay of 24 hours is kept; otherwise it is adjusted as follows: *republishing delay* $= \frac{load\ factor}{100} * 7 * 24$. The maximum republishing delay can thus be as long as 7 days.

The four most important message types for the route, publish, and search process are:

- `hello`: to check if the other peer is still alive and to inform the other peer about one's existence and the IP address and KAD ID.
- `route request/response(kid)`: to find peers that are closer to the KAD ID `kid`.
- `publish request/response`: to publish information.
- `search request/response(key)`: to search for information whose hash is `key`.

8.3 Crawling Peers with Blizzard

The three major challenges in crawling a peer-to-peer network are

- The time necessary to carry out a single crawl, which should be as small as possible to get a consistent view of the system.
- The frequency of the crawls, i.e., the time elapsing between two consecutive crawls should be small (no more than a few minutes) in order to achieve a high resolution for metrics such as session length.
- The duration of the crawl, which should be in the order of many months, to be able to correctly capture the longest session and inter-session lengths.

As mentioned earlier we have developed *Blizzard*, our own crawler for KAD, with the aim to crawl KAD very frequently and over a duration of several months. Our crawler logs for each peer P the time of the crawl, the IP address of P, the KAD ID of P, and whether or not P has responded to the crawler.

In a large peer-to-peer system such as KAD, peers are constantly joining and leaving, which makes it difficult to get a consistent view of the system. Therefore, the overall duration of a single crawl should be as short as possible. To speed up a single crawl, previous crawlers (such as the one developped by Stutzbach and Rejaie in [123]) were often distributed and ran simultaneously on multiple machines. However, we noticed that in a distributed crawl a lot of CPU time is used up for the synchronization between the different machines. To make our crawler run very fast, we decided to run *Blizzard* on a *single* machine and to keep all relevant information in main memory. The implementation of *Blizzard* is straightforward: It starts by contacting a seed peer run by us. Then it asks the seed peer for a set of peers to start with and uses a simple breadth first search and iterative queries. It queries the peers it already knows to discover new peers. For every peer returned, the crawler checks if this peer has already been discovered during this crawl. We use a hash table of already discovered peers that fits in main memory, which makes this test very efficient. After one crawl is completed, the results are written to disk.

At the beginning of each crawl, the number of new peers discovered grows exponentially before it approaches asymptotically the total number of peers in KAD. At some point the crawl needs to be stopped, otherwise the crawl accuracy decreases since new peers are joining the system all the time [121]. We choose to stop querying new peers when 99% of the peers discovered have been queried. We then wait for 30 more seconds for late replies before terminating the crawl.

Not all the peers discovered can be contacted directly by the crawler. Approximately half of the peers queried do not respond to the crawler. There are two main reasons why a peer does not respond to our queries: either the peer has left the system, or the peer is behind a NAT that blocks our query. For the crawler it is not possible to distinguish between these two cases.

The crawler is implemented as two asynchronous threads: One thread to send the `route requests(kid)` (Algorithm 9) and the other one to receive and parse the `route responses` (Algorithm 10). A list that contains all the peers discovered so far is used and maintained by both threads. The receiving thread adds the peers extracted from the `route responses(kid)` to the list, whereas the sending thread iterates over the list and sends 16 `route requests(kid)` to every peer in the list. The value of the KAD ID `kid` is different in each of the 16 `route requests`. Care is taken to assure that each value of `kid` falls in a different bucket of the peer's routing tree, which allows us to minimize the overlap between the sets of peers returned in the response.

Algorithm 9: send thread (is executed once per crawl)

Data: *peer*: struct{IP address, port number, `kid`}
Data: *shared list* Peers = list of *peer* elements
`/* the list of peers filled by the receive thread and worked on by the send thread */`
Data: *int* position = 0
`/* the position in the list up to which the peers have already been queried */`
Data: *list* ids = list of 16 properly chosen *kid* elements

```
1 Peers.add(seed); /* initialize the list with the seed peer */
2 while position < size(Peers) do
3     for i=1 to 16 do
4         destkid = Peers[position].kid ⊕ ids[i]; /* normalize the bucket to
              the peers position */
5         send route requests(destkid) to Peers[position];
6     position++;
```

Algorithm 10: receive thread (waits for the `route response` messages)

Data: *message* mess = `route response` message
Data: *peer*: struct{IP address, port number, `kid`}
Data: *shared list* Peers = list of *peer* elements
`/* the list shared with the send thread */`

```
1 while true do
2     wait for (mess = route response) message; foreach peer ∈ mess do
3         if peer ∉ Peers then
4             Peers.add(peer);
```

There are various pitfalls when crawling a peer-to-peer system for such a long duration, such as incomplete data due to crawler crashes, loss of network connectivity, or random failures due to temporary network instability. To address these problems, we run simultaneously two instances of our crawler, one at the University of Mannheim, Germany, connected to the German research network, and a second one at Institut Eurécom, France, connected to the French academic network. Running

two crawls in parallel turned out to be very useful: at some point, due to network problems, one instance of the crawler was seeing fewer peers than the other one. Also, occasionally, one of the two crawlers crashed.

The main difficulties while implementing the crawler were to

- parse the packets, since the protocol is not well documented and since the aMule implementation uses their own data types and byte order to build the packets.
- implement the function to look up if an <IP address, port> combination already exists. The first idea was to use a database, however, it turned out to be too slow. Since every entry only takes 100 bytes, it is possible to maintain the queue in main memory. However a adequate structure had to be found to answer the *contains* question. A hash map was the solution to this problem.

8.3.1 Data Cleaning

Hole stuffing

Crawling a representative subset happens periodically with a new crawl every five minutes, with the two crawlers at the Universität Mannheim and Eurécom being synchronized. A peer that replied to at least one of the two crawlers during round i is considered to be up at round i [1]. The snapshots obtained by both crawlers are not always identical. The difference in the number of peers discovered is sometimes in the order of 10%; most of the time the difference is less than 1%. Analyzing these differences reveals that the peers seen by one crawler but not by the other one are well distributed over all countries and the entire IP address space. We conclude that the temporarily seen large discrepancies up to 10% in the number of peers are due to local network restrictions, e.g., bandwidth shaping or overloaded routers, close to the site of one of the crawlers.

However, we realized that a peer that is up may occasionally be declared by both crawlers as not responding, i.e., considered as being down. One reason can be that the peer is overloaded and does not reply to our query. To validate this hypothesis we ran a KAD peer on an ADSL line: when neither the machine nor the peer application was heavily loaded it always responded to the crawlers. When the machine was loaded with heavy calculations the peer still responded. However when the KAD peer was loaded with a large number of simultaneous downloads it frequently did not respond to our crawlers.

Another reason can be that the path between the two crawlers and the peer is disrupted somewhere close to the peer. In both cases, the crawlers will not receive a response from the peer even when it is up and running. While it is not possible to

[1] As we can see in Algorithm 9 the crawler sends 16 route requests to each peer. A peer is considered alive if at least one route response is received by the crawler.

tell exactly why a peer is not answering, we implemented the following data cleaning rule that we consider "reasonable": When a peer P that has been reported up at round $i-1$ does not reply to either of the two crawlers during the next round i and then replies again during round $i+1$, then peer P will also be considered up at round i.

We refer to this filtering mechanism as eliminating one hole. One can of course generalize this approach to eliminating i holes, which means considering a peer that responded during crawl k, then did not respond for up to i consecutive crawls before responding again, as being continuously up from crawl k to crawl $k+i+1$.

Since we have no answer as to what data cleaning technique is the most appropriate, we ran different experiments where we eliminated i holes, with $i \in \{0, 1, 2, 3, 5\}$. The resulting Cumulative Distribution Functions (CDFs) of the session times are shown in Figure 8.2, and the first two moments of the session times in Table 8.1. Of course, the bigger the holes we eliminate the larger the mean session times. However, it is important that independent of the number of holes eliminated, the session times could always be a perfect fit using a Weibull distribution, which is described by two parameters, referred to as scale and shape. We will come back to this in section 9.2.6.

For the rest of this section we will use the data with 1 hole eliminated.

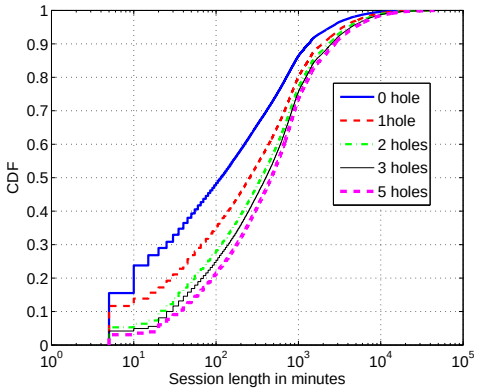


Figure 8.2: CDF of the session times (1, 2, 3, and 5 hole(s): after data cleaning, 0 hole: raw data without data cleaning).

	Weibull		Session times	
Eliminating	Scale	shape	mean	std dev.
Raw data	55.91	0.52	113.73	297.41
1 hole	97.62	0.56	169.21	405.26
2 holes	129.53	0.61	199.62	455.9
3 holes	144.28	0.63	215.8	486.2
5 holes	165.4	0.65	238.38	525.25

Table 8.1: Session characteristics before and after data cleaning. Eliminating i holes means that we consider a peer as *connected* even if it does not respond during i consecutive crawls.

Artifacts

There are two different types of artifacts one can notice in the crawl results:

- There are IP addresses that host many KAD IDs (cf. Section 8.2).

IP address	Country	# of KAD IDs
219.144.16.135	CN	197
83.35.252.57	ES	190
84.185.82.103	DE	189
88.3.70.67	ES	177
213.197.129.54	LT	144
87.203.219.36	GR	135
200.203.111.117	BR	125
221.238.129.241	CN	113
201.15.49.126	BR	112
86.100.3.250	LT	110

Table 8.2: Many KAD IDs running on one IP address.

In some cases all the peers have the same hash value on successive port numbers. These are single entities analyzing the KAD network or trying to gain higher download speeds. One can see that all these peers are run indeed on one single physical machine since the measured RTTs are very close to each other (Table 8.3).

Recently, Pietrzyk et al. [91] monitored a population of about 20,000 ADSL clients belonging to a major ISP in France. About 20% of the peers change their KAD ID regulary, using a new KAD ID for every new session. Sometimes running sessions are even interrupted only to immediately reconnect using a new KAD ID. This analysis confirms our findings crawling KAD.

IP address	port	Country	KAD ID	rtt
83.35.252.57	50837	ES	9dcccb9d	543
83.35.252.57	50666	ES	9dcccb9d	361
83.35.252.57	50265	ES	9dcccb9d	613
83.35.252.57	50749	ES	9dcccb9d	513
83.35.252.57	50875	ES	9dcccb9d	545
83.35.252.57	50670	ES	9dcccb9d	457
83.35.252.57	50366	ES	9dcccb9d	571
83.35.252.57	50535	ES	9dcccb9d	567
83.35.252.57	50831	ES	9dcccb9d	503
83.35.252.57	50842	ES	9dcccb9d	476

Table 8.3: A single IP address and KAD ID but many different ports.

In other cases, the peers have different random hash values and non-consecutive port numbers. These are clients sharing the same public IP address behind a NAT. Since we deal with different physical machines, also the

measured RTTs are very different from each other. Peers that did not reply are marked with -1 (Table 8.4).

IP address	port	Country	KAD ID	rtt
221.238.129.241	44457	CN	68d0ad00	-1
221.238.129.241	30709	CN	3db7742b	324336
221.238.129.241	37411	CN	d40e7f35	-1
221.238.129.241	34681	CN	22ff4d0c	416
221.238.129.241	19089	CN	e05d2942	-1
221.238.129.241	12855	CN	dd344f05	1925
221.238.129.241	7904	CN	0be4b864	554
221.238.129.241	64076	CN	363bfc47	422

Table 8.4: Peers behind a NAT: One IP address and many ports and KAD IDs.

- The same hash is used by many different peers with different IP addresses.

For the evaluation we filtered out all these artifacts.

8.4 Spying for Content with Mistral

In this section we explain how to get an overview of the content in KAD using our content spy *Mistral*. It is based on the same principle as the *sybil attack* [28, 31, 33]. We introduce a large number of our own peers, the *sybil*s, into the network, all controlled by one machine. Positioned in a strategic way in the KAD space but physically all running on the same machine, the sybils can gain control over a fraction of the network or even over the whole network. The fact that all sybils run on the same machine has the advantage that data collection is much easier. The sybils can monitor the traffic or even abuse the KAD protocol: routing requests may be forwarded to wrong directions or rerouted to entities under our control.

We insert a large number of sybil peers into the network and propagate information about them into the routing tables of the real peers. The first step involves learning about the peers present in the network; we thus crawl the peers with *Blizzard*. A query message asking for peers in a certain region of the KAD space includes the KAD ID of the asking peer; thus, we send it only after we got the KAD ID by crawling. The fact that the KAD ID of the receiver has to be known to query a peer is actually the only security feature in the KAD protocol. The query thus publishes the IP address of the machine running *Mistral*.

Note that so far the KAD IDs of the sybils are not yet known to other peers. In a second step, we send `hello` messages to the peers we have learned about during the *Blizzard* crawl. A `hello` message includes the KAD ID of the sender; we can choose it freely. The first 24 bit are chosen at random, while the 96 remaining bits are fixed. This is a signature of our sybils, allowing us not only to trace them by their IP address but also by their KAD ID.

The routing queries reaching the sybils are always answered with other sybils. The returned KAD ID is closer to the target included in the query than the receiver of the query, thus the querying peer has always the impression of approaching the target. Once the requester gets close enough to the target, it queries a sybil for the content itself and not for any closer peers. Our sybil stores the search request and returns a fake source entry. This source entry points to our machine. As a consequence, the real peer tries to start to download which is not successful.

Not only routing and search requests are hitting our machine but also requests to publish. As stated above, these are especially interesting since they are much more frequent than search requests. Whereas search requests are always launched by a human, publish requests are automatically and regularly launched by the KAD clients. Also, the publish information is richer than the search requests: it includes the full file name, the KAD ID of the source and a significant amount of metadata on the file. As explained above, the filename is tokenized and published on the hash of *each* of its tokens (keywords). The answer to a publish request is the load of the peer addressed. We always answer with a very low load, thus we attract more and more publish requests to our sybils.

An 8-bit zone contains the peers whose KAD ID agrees in the first 8 bits, thus each zone can theoretically contain 2^{120} hash values. We actually observe between 12,000 and 25,000 peers per zone. The entire KAD network contains 256 8-bit zones and between 3 and 5 million peers. It is possible to spy on one zone of the KAD network only by restricting the returned KAD IDs to a certain prefix. Into a zone we can insert 65,356 distinct sybils to be sure to catch at least one of the ten publish messages for a keyword or a source and at least one of the three search messages that are sent per user-initiated search.

To resume, the *Mistral* spy works as follows:

- First, crawl a zone $\mathcal{Z}$ of the KAD ID space using our crawler *Blizzard* to learn about the peers $\mathcal{P}$ currently online whose KAD IDs are in $\mathcal{Z}$.
- Then, send `hello requests` to the peers $\mathcal{P}$ in order to "poison" their routing tables with entries that point to our *sybil*s. The peers that receive a `hello request` will add the *sybil* to their routing table if the corresponding bucket of the routing table is not filled.
- Later, when a `route request(kid)` initiated by regular peer P reaches a *sybil* that request will be answered with a set of *sybil*s whose KAD IDs are closer to the target in case the `kid` falls into the zone $\mathcal{Z}$, and will be ignored otherwise.

  This way, P has the impression of approaching the target. Once P is "close enough" to the target KAD ID, it will initiate a `publish request` or `search request` also destined to one of our *sybil* peers. Therefore, for any `route request` that reaches one of our *sybil* peers we can be sure that the follow-up `publish request` or `search request` will also end-up on the same *sybil*.
- Store the content of all the requests received in a database for later evaluation.

A key is published ten times and for a content search three parallel search requests are issued. For our spy scheme to work as intended, the optimum would be to attract exactly one copy of every search or publish request. In this way, publish and search requests would also "terminate" on regular peers that would correctly execute them, avoiding any disruption of KAD due to our spy. There are two parameters to control the level of intrusiveness: The number of *sybil*s placed in a zone and the rate at which *sybil*s are announced to regular peers.

Content Crawler Another possibility to learn about the content in KAD is to search for *all* keywords. First of all one needs to know all these keywords. However, it is impossible to know all keywords used. We implemented a content crawler that searches for a given *seed* keyword using an exhaustive search procedure. First the tolerance zone around the keyword is crawled for peers, then all these peers are queried for the keyword. The file names that are returned to these queries are parsed and the keywords learned are again searched for. This method works very well, we where able to obtain 4 million keywords and tens of millions of files in the period of some days. However it would take months to crawl the entire content in KAD with the available resources we have.

8.5 Azureus

The DHT Kademlia [77] is also implemented in AZUREUS [6]. The user base is about 1 million concurrent users, which is smaller compared to eMule and aMule 9.1.1. Azureus is currently one of the most popular clients for BitTorrent. For a given file, the protocol embodies three main roles: a tracker, seeders, and peers. Trackers are typically web servers: seeders and peers are transient clients. The initial seeder is the source of the file. It divides the file into small pieces, creates a metadata description of the file and sends this description to the tracker. Peers discover this file description through some out-of-band mechanism (e.g., a web page) and then begin looking for pieces of the file. Peers contact the tracker to bootstrap their knowledge of other peers and seeds. The tracker returns a randomized subset of other peers and seeds. Initially, only the initial seeder has pieces, but soon peers are able to exchange missing pieces with each other, typically using a tit-for-tat scheme. Once a peer acquires all of the pieces for a file, it becomes a new seeder. The set of clients actively sharing a file is called a swarm. In AZUREUS, file descriptors and other metadata are stored in a Kademlia-style DHT, in which all clients participate.

8.6 Vivaldi Network Coordinates

Internet coordinate systems allow a host to predict the round trip times to other hosts without actually measuring them. A coordinate system can be used to decide

from which peer in a peer-to-peer network to fetch a data item from. Explicit measurements are often unattractive because the cost of measurement can outweigh the benefits of exploiting proximity information. Coordinates are assigned to hosts such that the distance computed using coordinates predicts the RTT between these hosts. Simulation-based systems map nodes and latencies into a physical system whose minimum energy state determines the node coordinates.

Vivaldi [26] calculates the coordinates as the solution to a spring relaxation problem. The system envisions a spring between each pair of nodes, with the resting position of the spring equaling the network latency between the pair. Each node successively updates and refines its own position by taking into account newly reported RTT measurements toward its communication partners. Since this information is piggybacked on other network messages, e.g., route requests, no additional messages are sent through the network. In other words, nodes allow themselves to be pulled or pushed by a connected spring. A network embedding with a minimum error is found as the low-energy state of the spring system.

Vivaldi uses Euclidean coordinates of d dimensions augmented with a height value x_h: $x = x_1, \ldots, x_d, x_h$. The coordinates without the height can be seen as a high speed Internet core to which are end users attached. The last mile that may suffer from queuing delays due to large buffers, e.g., in the case of ADSL, is represented by the height value. Without using the height, the coordinate space would be distorted since it is possible to measure latencies in the order of seconds between peers in the same country, which is more than the propagation time needed to make the tour of the globe (≈500 ms). A packet sent from one node to another must travel the source node's height, then travel in the Euclidean space, then travel the destination node's height. Even if the two nodes have the same height, the distance between them is their Euclidean distance plus the two heights. This is the fundamental difference between height vectors and adding a dimension to the Euclidean space. Intuitively, packet transmission can only be done in the core, not above it. To calculate the distance between two nodes x and y, first the distance of their Euclidean coordinates is calculated, then the heights of both nodes are added:

$$dist(x, y) = \sqrt{\sum_{i=1}^{d} (x_i - y_i)^2} + x_h + y_h.$$

In a normal Euclidean space, a node that finds itself too close to another node will move away from the other node. A node that finds itself too close to nodes on all sides has nowhere to go: the spring forces cancel out, and it remains where it is. In the height vector system, the forces cancel out in the Euclidean plane, but the height forces reinforce each other, pushing the node up away from the Euclidean plane. Similarly, a node that finds itself too far away from other nodes on all sides will move down, closer to the plane.

CHAPTER 9
Peer behavior in KAD

In this chapter, we focus on a single peer-to-peer system, namely KAD, which is the publishing and search network of eDonkey. We want to characterize KAD in terms of metrics such as arrival/departure process of peers, session and inter-session lengths, availability, and lifetime.

To obtain the relevant raw data we decided to "crawl" KAD. Each crawl gives a snapshot of the peers active at that instant.

While peer-to-peer systems have been explored previously using a crawler, the duration of these crawls was limited to a few days at best. We were able to crawl KAD for almost six months at a frequency of one crawl every five minutes, which makes a total of 51,552 snapshots. We obtain a number of original results such as:

- Session lengths are heavy–tailed, with sessions lasting as long as 78 days and are the best characterized by a Weibull distribution, with shape parameter $k < 1$. One property of Weibull distributed session lengths is that a peer that has so far been up for t units of time will – in expectation – remain up for a duration that is in the order of $O(t^{1-k})$. We can exploit this fact to use the past uptime in order to predict the remaining uptime.

- For many peers, the amount of time a peer is connected per day, called daily availability, varies a lot from one day to the next. This makes it difficult to predict daily availability

- Contrary to what was known up to now, KAD IDs are not persistent and can change as frequently as once per session. By using a subset of peers with static IP addresses we can also show that the end-user lifetime is significantly longer than KAD ID lifetime with 50% of the peers participating in KAD for six month or more.

- When classifying peers according to their geographic origin, the peers from China make about 25% of all peers seen at any point of time and Europe is the continent where KAD is most popular. We also see a big difference between

peers in China and Europe with respect to some of the key metrics such as session length or daily availability.

The following two sections present the results of our crawl. Section 9.1 will provide general information such as the number of KAD users and their geographic distribution and will also discuss KAD ID aliasing and its implications [112, 117]. Section 9.2 will focus on statistics related to session times that are very relevant for the optimization of certain design parameters of KAD such as the republishing interval and metrics that characterize the daily usage behavior of KAD clients [114].

9.1 Global View of KAD

In this section, we will present results obtained via a full crawl of KAD, such as the total number of users, the geographic distribution of the users, and the distribution of the KAD IDs over the hash space. We will and compare these results where appropriate to the ones obtained via the zone crawl. Moreover we will characterize the fact of IP address aliasing and KAD ID aliasing.

9.1.1 Full Crawl

The speed of *Blizzard* allows us to crawl the entire KAD ID space, which was never done before. Such a **full crawl** of KAD takes about 8 minutes. The first million different peers are identified in about 10 seconds, the second million in 50 seconds, thereafter the speed of discovery decreases drastically since most peers returned in the `route response` messages have already be seen before during the same crawl (Figure 9.1). A full crawl of KAD produces about 3 GBytes of inbound and 3 GBytes of outbound traffic.

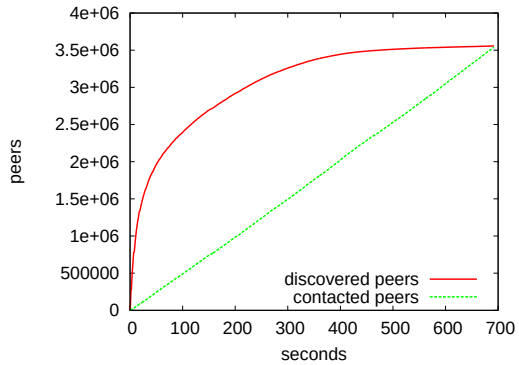


Figure 9.1: The number of discovered peers approaches asymptotically the total number of peers in the network.

A full crawl was done once a day from March 20, 2007 to May 25, 2008.

During each full crawl, we found between 3 and 4.3 million different peers. Between 1.5 to 2 million of these can be *directly contacted* by our crawler. The other peers either have left the system or are located behind NATs or firewalls. In the rest of this chapter we will only report statistics on the peers that our crawler could contact *directly*.

As we can see in Figure 9.2 the number of peers seen varies according to a diurnal and a weekly patterns and reaches its peak during the weekend, where the population is about 10% higher.

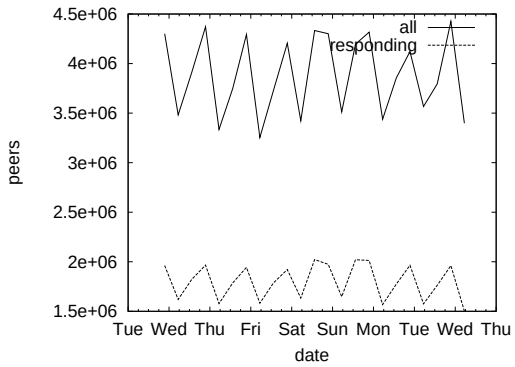


Figure 9.2: The number of KAD peers available in **entire** KAD ID space depending on the time of day.

In Figure 9.3, we plot the distribution of the percentage of peers seen per country on August 30, 2006. Using the Maxmind database [76] to resolve IP addresses to countries and ISPs. The continent with the highest percentage of peers is Europe (Spain, France, Italy and Germany), while the country with the largest number of peers is China. Less than 15% of all peers are located in America (US, Canada, and South America). We can also see that the geographic distribution of the peers obtained with the two zone crawls of an 8-bit zone each is very close to the result obtained with the full crawl, which is to be expected since the KAD IDs are chosen at random.

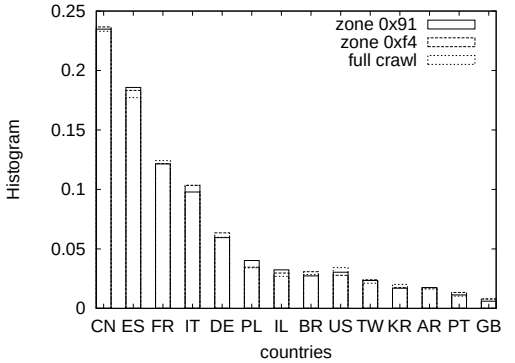


Figure 9.3: Histogram of geographic distribution of peers seen.

In fact, we see from the results of the full crawl that the peers are uniformly distributed over the hash space, except for some outliers (Figure 9.4). All the outliers are due to modified KAD clients that use the same KAD ID and are always limited to

one country (Korea, Spain, Israel, China, Argentina). The outlier in zone `0xe1` is a modified client used in Israel, for which we counted 25069 instances.

The other outliers are modified clients (mods) as well, they are always limited to one country to one language zone. The people using mods are sort free-riders. The overhead is divided among them, but they can use the network a anyone else. However if too many people are using mods the performance decreases, since the people using the same hash can not see one another, and therefore can not download one from another. Furthermore the DHT gets less stable.

We also saw that the US-based company named Media Defender is running up to 250,000 instances of KAD, with KAD IDs that systematically cover the entire KAD ID space. By carrying out a so called Sybil attack 8.4, Media Defender is able to closely monitor all publish and search activities of all peers in KAD. We have "filtered out" these "anomalies" from our trace data since we are interested in characterizing the behavior of ordinary KAD peers.

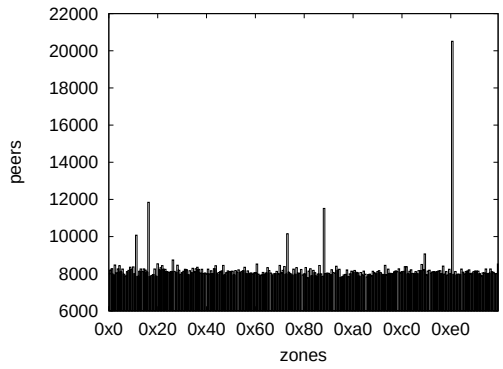


Figure 9.4: The distribution of the peers over the hash space. The 256 8-bit zones on the x-axis go from `0x00` to `0xff`.

Since the full crawl was quite expensive in terms of resources, we will extensively rely on the zone crawl to obtain much of the relevant information about KAD.

9.1.2 Zone Crawl

A full crawl generates an extremely high amount of trace data and of network traffic (with peak data rates close to 100 Mbit/sec). Also carrying out just 3 crawls per day is not really sufficient to capture the dynamics of KAD peers at short timescales, which is needed to measure e.g. session times. For this reason, we decided to carry out a **zone crawl** on a 8-bit zone, where we try to find all active peers whose KAD ID have the same 8 high-order bits. Such a zone crawl, that explores one 256-th of the entire KAD ID space, takes about 2 seconds and generates 25MB of traffic (Figure 9.5). Note that only after the first replies arrive (after 500-600 msecs) the exponential grow in the number of discovered peers can start.

The output of a zone crawl looks like this:

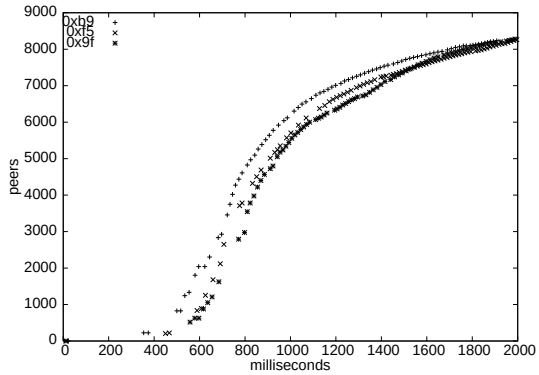


Figure 9.5: The time needed to crawl one zone, a $\frac{1}{256}$ part of the total KAD network.

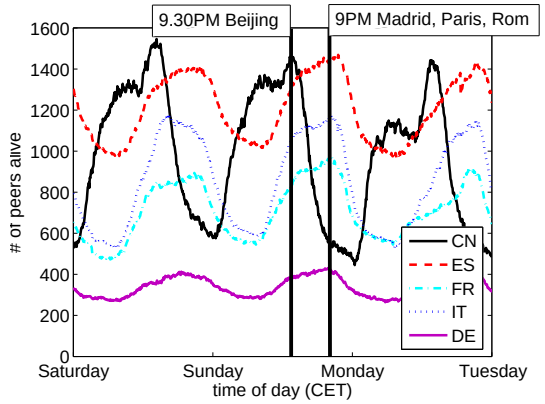
```
crawl zone  0xa1
time 1855ms
packets received: 71757 unique IP/port: 9028
packets sent: 131968
traffic in: 16.15 MB
traffic out: 9.69 MB
5134 of 9028 are alive
```

The high resolution and long duration zone crawl from September 23, 2006 to March 20, 2007 (179 days) allowed us to collect 51,552 snapshots (one every 5 minutes) of a subset of all KAD peers. In this section, we will use the results of a full crawl to validate that the subset of KAD peers captured via a zone crawl is indeed a representative sample of the peers in KAD.

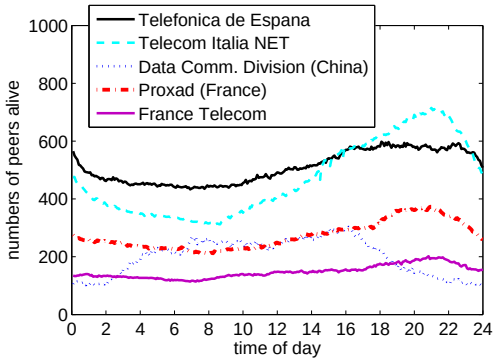
In Figure 9.6(a), we plot the number of peers seen that originate from China and some European countries. The number of peers in each country follows a diurnal pattern, with a peak around 9 PM local time. The eight hour time shift between Europe and China is clearly visible.

The most important providers do all come from the countries accommodate most of the KAD peers. As for the countries of origin also for the ISPs a diurnal pattern can be detected (Figure 9.6(b)).

Table 9.1 summarizes the basic findings on the zone crawl. The peers seen came from 168 different countries and 2,384 providers. For the KAD IDs seen the 1st day of our zone crawl, we observe that about one third of the peers come from Europe and about one fourth from China. If we compare the **lifetime** of the peers, which is defined as the difference between the time a given KAD ID was seen the last time and the time this KAD ID was seen the first time, we notice that the lifetime of peers in China is much smaller than the one for peers in the other countries. More than half of the peers in China were seen for the duration of only *one* session. We will come back to this point in subsection 9.1.3.



(a) Peers online according to country of origin.



(b) Number of peers only for the most important providers.

Figure 9.6: Peer arrivals between two crawls during the first week.

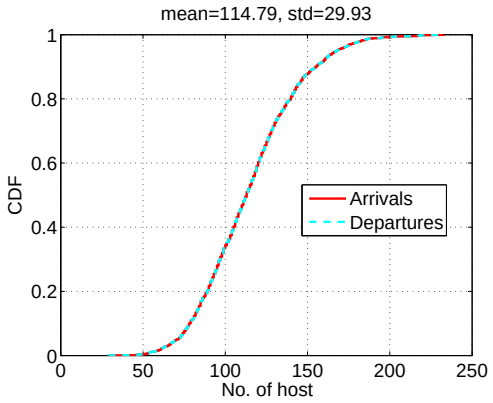
	Total	China	Europe	Rest
Different KAD IDs	400,278	231,924	59,520	108,834
Different IP addresses	3,228,890	875,241	1,060,848	1,292,801
KAD IDs seen for a single session	174,318	131,469	11,644	31,205
KAD IDs with LT $\leq$ 1 day	242,487	183,838	15,514	43,135
KAD IDs seen for the first time on				
- 1st crawl	5,670	455	2,879	2,336
- 1st day	18,549	4,535	6,686	7,328
- 60th day	1,893	1,083	259	551
KAD IDs seen for the first time on 1st day				
- with LT $\leq$ 1 day	2,407	1,568	286	553
- 1 day $<$ LT $\leq$ 1 week	1,368	497	393	478
- 1 week $<$ LT $\leq$ 1 month	2,735	791	944	1,000
- LT $>$ 1 month	12,039	1,679	5,063	5,297
- LT $>$ 3 months	8,423	936	3,679	3,808
avg. of the median session time per peer (minutes)	165	103	326	210
avg. of the median-inter session time per peer (minutes)	1,341	586	2,825	2,136

Table 9.1: Key facts about the zone crawl spanning 179 days organized by country of origin (LT=Lifetime).

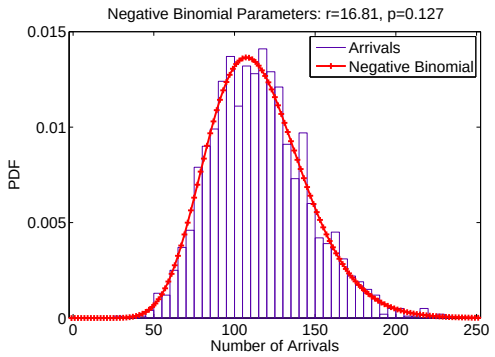
Arrivals and Departures

Since we crawl the same zone in KAD once every 5 minutes, we can determine the number of peers that join and leave between two consecutive crawls. Knowing the arrival rate of peers is useful since it allows us to model the load in KAD due to newly joining peers. Each time a peer joins, it first contacts other peers for information to populate its routing table, before it publishes the keywords and source keys for all the files it will share.

In Fig. 9.7(a) we depict the CDF of the number of peers that arrive and that depart between two consecutive zone crawls. We see that the distributions for arrivals and departures are the same. This is to be expected, since we observe the system in "steady state": in this case, the system should behave like $G/G/\infty$, for which, according to Little's Law, the arrival rate is equal to the departure rate [81]. The arrival process is very well described by a Negative Binomial distribution (Figure 9.7(b)).



(a) CDF of the number of arrivals and departures.



(b) PDF of the number of arrivals

Figure 9.7: Peer arrivals between two crawls during the first week.

Given that KAD IDs are uniformly distributed, we can estimate the total number of peers in KAD by simply counting the number of peers in a zone and multiplying this value by the number of zones (256 zones). Using Chernoff Bounds (see [82] Chapter 4) we tightly bound the estimation error.

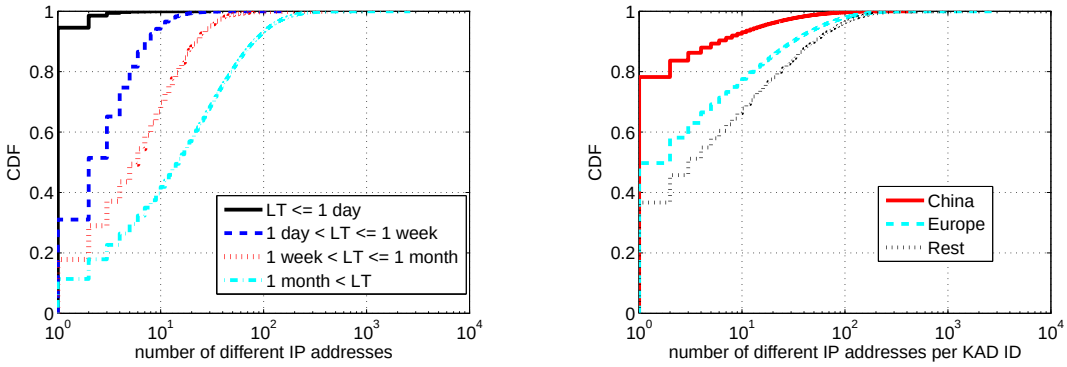
Let $N(t)_{part}$ be the number of peers counted during a zone crawl of an 8–bit zone at time t and $\hat{N}(t) := 256 * N(t)_{part}$ the estimate for the total number of peers in the KAD system. The true value $N(t)$ for total number of peers at time t is very close to the estimate $\hat{N}(t)$, with high probability. More precisely: $Prob[|N(t) - \hat{N}(t)| < 45000] \geq 0.99$, which means that our estimate $\hat{N}(t)$ has most likely an error of less than 3% for a total population of at least 1.5 million peers.

9.1.3 Aliasing

IP Address Aliasing

It has been known for several years [8, 64] that many peers frequently get assigned new IP addresses, which is referred to as **IP address aliasing**. For instance, we know that some ISPs in France change the IP address of their ADSL customers approximately every 24 hours, while others assign static IP addresses to their clients. We observed a total of 400,278 distinct KAD IDs and 3,228,890 different IP addresses (see Table 9.1). In Europe, a peer has on average about 18 IP addresses, whereas in China the number is 4 IP addresses per peer. About 80% of the peers in China have only

one IP address since their lifetime is much shorter than the lifetime of peers in other parts of the world. We saw that the number of different IP addresses per peer is strongly correlated with the peer lifetime (Figure 9.8).



(a) Peers organized according their lifetime.

(b) Peers organized according their origin.

Figure 9.8: The number of different IP addresses reported per KAD ID.

KAD ID Aliasing

Up to now it was assumed that KAD IDs are persistent, i.e., the same end-user of KAD *permanently* keeps the same KAD ID across all its sessions. As it turns out, this is not true. We refer to the fact that KAD IDs are non-persistent as **KAD ID aliasing**.

We see in our zone crawl approx. 2,000 new KAD IDs a day, which means that for the entire KAD system the number of new KAD IDs per day is around 500,000. If we extrapolate, this makes about 180 Million KAD IDs a year. It is hard to believe that there exist such a large number of different end-users of KAD.

Figure 9.9 reports the number of *new* KAD IDs per day. i.e., KAD IDs seen for the first time, according to country of origin. More than 50% of the new KAD IDs are from peers in China, which is more than one order of magnitude greater than the number of new KAD IDs seen for any other country such as Spain, France, or Germany.

We were curious to find out whether it is plausible that many end-users really stop using KAD after one session, or whether the same users come back with a different KAD ID. To investigate KAD ID aliasing, we need to look for peers with *static IP addresses*, which we can track for *non-persistent* KAD IDs. We know that, for instance in France, one of the ADSL providers (Proxad) assigns static IP addresses to customers who are located in areas where the service offer is completely "un-bundled".

Our hypothesis is that a peer that keeps the same IP address and port number for 10 days is assigned a static IP address. Therefore, we take the logs of the two full crawls (cf. Section 9.1.1) of March 20, 2007 and March 30, 2007 and extract the 140,834 peers that have the same IP address, port number and KAD ID in both crawl logs. IP addresses running more than one KAD ID are filtered out. This way we exclude all

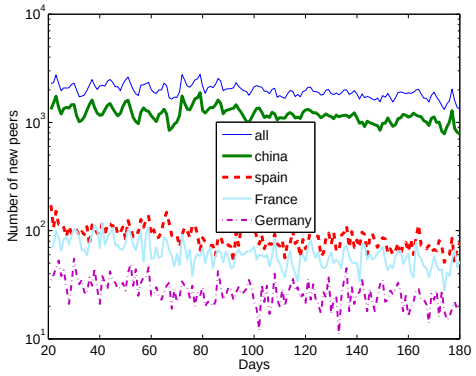


Figure 9.9: New KAD IDs according to country of origin.

users having dynamically assigned IP addresses, moreover we exclude all users with static IP addresses who were not online on March 20 and March 30. We call this set of peers a **pivot set**.

Since this heuristic is very strict, the number of users with static IP addresses is underestimated. However the pivot set still contains enough peers to make statistically meaningful statements. 32% of the peers in the pivot set come from Spain, 18% from France, 5% from Poland and Italy, 4% from the US and Taiwan, and 3% from Israel and Argentina.

We then take the logs of the full crawls starting March 31st, 2007 to look for peers in the pivot set that have *changed their* KAD ID.

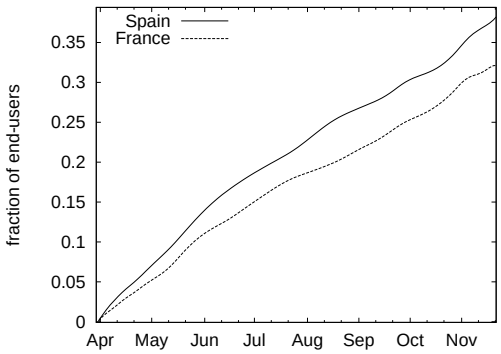


Figure 9.10: The fraction of peers in the pivot set that changed their KAD ID at least once.

In Fig. 9.10, we plot the fraction of peers from the pivot set that change their KAD ID at least once. Since we perform a full crawl only once a day, we are not able to estimate the rate of change of the KAD IDs. Instead, we can only detect which peers have changed their KAD ID. We see that a significant fraction of end-users in different countries change their KAD ID over time. After seven months, more than one third of the end-users in Spain and France changed their KAD ID at least once.

A very recent study confirmed that KAD ID aliasing is quite common. Pietrzyk et al. [91] monitored a population of about 20,000 ADSL clients in France for ten days.

About 20% of the peers change their KAD ID for every new session and some peers change it even during a session. In comparison to clients that do not change their KAD ID, these peers have longer session times, whereas the amount of files they share is significantly smaller. It seems that peers who frequently change their KAD ID do so in order to improve their anonymity.

Implications of KAD ID Aliasing The fact that the KAD ID assigned may be non-persistent obliges us to distinguish between a peer and an end-user:

- A **peer** is an instance of KAD identified by a fixed KAD ID.
- An **end-user** is a physical person that launches a peer to participate in KAD. The same end-user can, at different times, participate in KAD via different peers.

When KAD ID aliasing occurs, it is not really possible to characterize the lifetime of *end-users* tracking KAD ID, as compared to the lifetime of peers. All we can extract from our crawl data is the lifetime of *peers*, which provides us with a lower bound on the lifetime of end-users. We will see in section 9.2.6 how we can use the peers in the pivot set to estimate the lifetime of end-users.

9.2 Peer View

In this section, we will present metrics that describe the behavior of individual peers, such as lifetime, session and inter-session time, residual uptime, and daily availability using the observations made with our 179-day zone crawl. To estimate the end-user lifetime we also make use of the data obtained via the full crawl.

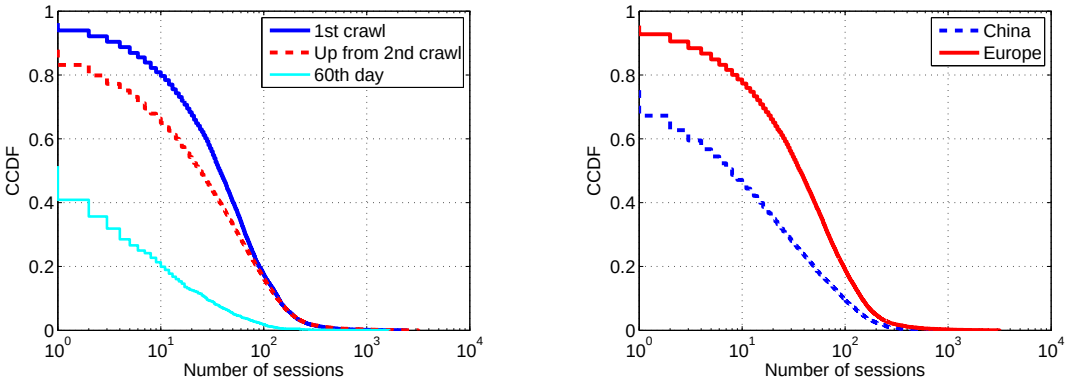
Using these metrics we will compare the peer behavior of different countries. Knowing the session statistics allows us (i) to validate implementation choices of KAD and (ii) to make suggestions on how to improve the efficiency of KAD (cf. Section 9.4).

We will mainly focus on the peers that were first seen on the 1st day of our crawl, since we could observe them for the longest period of time. For reference, we will occasionally compare the results with those obtained for the peers seen the first time on day 60. We have chosen day 60, since the largest inter-session times observed very rarely exceed 60 days, which allows us to assume that the peers we see for the first time on day 60 have most likely newly joined the system.

9.2.1 Session Statistics

Most of the peers will not be online, i.e., connected to KAD, all the time. By crawling KAD every five minutes, we can determine precisely for each peer k the instances $t_1^j(k), ..., t_n^j(k)$ when k joined and the instances $t_1^l(k), ..., t_n^l(k)$ when k has left KAD. We define the **session length** as the time a peer was present in the system without

any interruption, i.e., $t_i^l(k) - t_i^j(k)$ for $i \in \{1, ..., m\}$. For the peers that were online on the first crawl, we did not consider the first session, since we can not know when it began. Analogously, we did not consider the sessions that were still ongoing during our last crawl.



(a) (b) China vs Europe: Peers of the first day.

Figure 9.11: CCDF of the number of sessions per KAD ID.

In figure 9.11 the number of sessions per KAD ID is plotted. In figure 9.11(a), we compare the distributions of the number of sessions of peers seen in the first crawl, the first day up from second crawl and in the 60th day. Note that the peers seen in the 60th day were observed only 120 days wile the peers of the two other set are observed for 180 days.
60% of peers seen in the 60th day have only one session time. This percentage fall down to 18% for the set Up from second crawl and to less than 5% for the peer seen in the first crawl. (As we observe the 60th day peer for 120 days, this observation is not influenced by the difference of duration of the crawl). This observation can be explained by the fact that more than 90% of the peers of the 60th day are Chinese. This our belief of the existence of a modified KAD client in china. Among the peers seen on the first day, 20% have more than 100 sessions.

Figure 9.11(b) shows the difference between the CCDF of the number of session between peers coming from China and peers coming from Europe. First, we note that the distribution of European peers is very similar to the one of peers seen in the first crawl. Which mean that the most stable peers are from Europe.
30% of Chinese peers have only one session time compared to 5% for the European peers.
30% of peers in first day are Chinese, while in the first crawl they were only 8%.

The session length of the peers seen in the *first* crawl is about twice that of the peers seen for the first time during later crawls of day 1 (Fig. 9.12). When we crawl KAD for the first time, we have a much higher chance of seeing peers that are connected "most of the time" than peers that are connected from time to time and only for short periods. This means that a single crawl of the system cannot give a representative picture of the characteristics of the peers: Instead, we need to sample the system many times.

For the peers seen in the first crawl, we observe session times (in minutes) with a mean = 670, standard deviation = 1741 and median = 155. For the peers seen during the remaining crawls on the first day these values are only about half as large with mean = 266, standard deviation = 671 and median = 75. In both cases, the coefficient of variation, which is defined as the ratio between standard deviation and mean, which characterizes the "variability" of a distribution, is between 2 and 3.

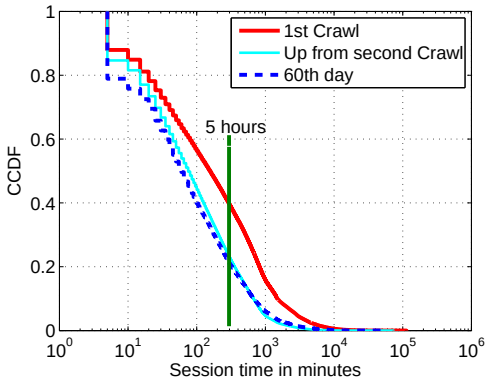
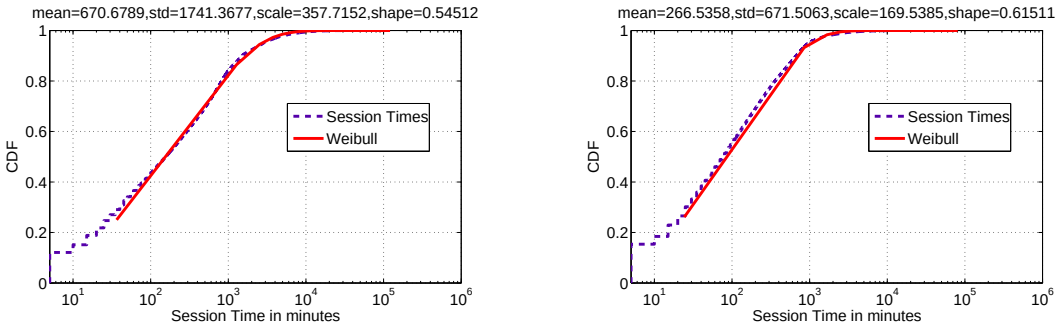


Figure 9.12: CCDF of the session lengths per KAD ID for different peer sets.

Weibull fit of the session time distribution

The empirical distribution of the session length exhibits a considerable tail. At least 0.1% of the sessions are longer than 1 week and the longest session observed is 78 days. We did a distribution fitting for the session times and found that the Weibull distribution provides a very good fit as we can see in Figure 9.14 (See Table 8.1 for Weibull parameters).

Using only the session length samples larger than 15 minutes, the fit passes the Kolmogorov-Smirnov (goodness of fit) test. However, for the small session lengths of 5, 10, or 15 minutes, the fit is not good due to the too large granularity of time between two crawls (5 minutes).



(a) First crawl (b) Up from 2nd crawl

Figure 9.13: Weibull fit of the session time distribution of the peers seen first day.

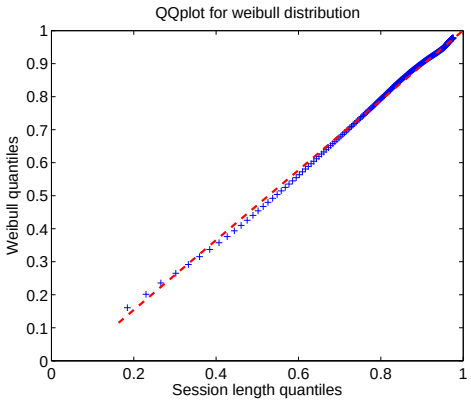


Figure 9.14: QQplot of Session length distribution versus Weibull distribution for peers seen in the first crawl.

The Weibull distribution has two parameters $k > 0$ (*shape*) and $\lambda > 0$ (*scale*). The Weibull distribution with $k < 1$ is part of the class of the so-called sub-exponential distributions, for which the tail decreases more slowly than any exponential tail [43]. Sub-exponential distributions are a subclass of the class of heavy-tailed distributions [12]. This implies that knowing the past (uptime) of a peer allows us to predict the future (residual uptime). More formally, if S denotes the session length then the expected residual uptime is $\mathbb{E}[S - t|S > t] \sim O(t^{1-k})$, i.e., it grows *sub-linearly*. For comparison: if S where Pareto distributed, the growth of its residual uptime would be linear, i.e., $O(t)$.

9.2.2 Remaining Uptime

Figure 9.15 shows the expected residual uptime for the scale and shape values that describe the session length of peers seen in the first crawl. There is a nice fit between the empirical values and the interpolation using a function whose growth is $O(t^{1-k})$. We see that for small observed uptime values the remaining expected uptime values are considerable: A peer that has been up for 1,000 minutes will have a remaining expected uptime of 1,500 minutes. Based on their data set, which did not contain any sessions longer than 1 day, Stutzbach and Rejaie [123] concluded that KAD sessions times could be fit either by a log-normal or a Weibull distribution. Our crawl, which allowed us to observe sessions that lasted several weeks, confirms this point.

The eMule and aMule implementations of KAD only publish on peers that have been up for *at least 2 hours*. Source keys will expire after 5 hours and keyword keys after 24 hours. We may ask whether selecting a peer that has been up for at least 2 hours will increase the chances that this peer will be up for another 5 or 24 hours.

In Fig. 9.16, we plot the remaining uptime of peers given that they have already been up 5 minutes, 1h, 2h, or 8h. For this analysis we choose the set of peers seen in the first crawl since this is the view a joining peer has of the system. We see that a higher

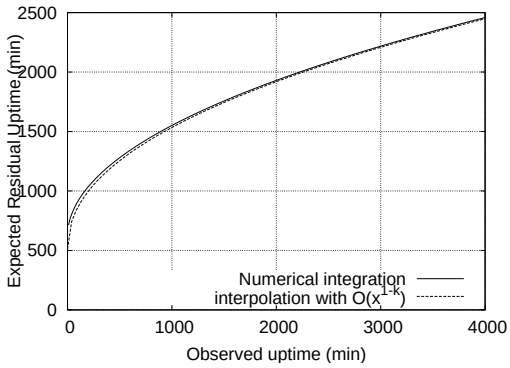


Figure 9.15: Expected residual uptime for $k = 0.54, \lambda = 357$ (for peers seen in the first crawl).

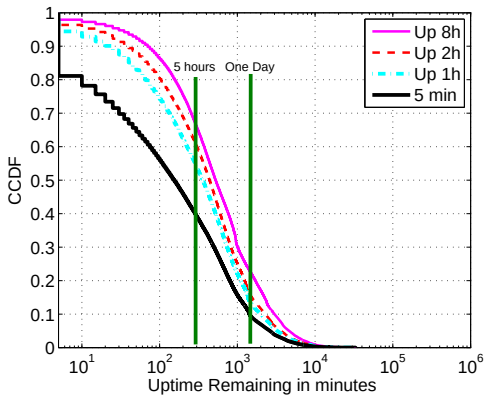


Figure 9.16: CCDF of the remaining uptime of peers, given the uptime so far, for peers seen in the first crawl.

uptime translates into a higher remaining uptime. This means that the minimum age-based peer selection as implemented in eMule and aMule is a sensible policy.

Only about 20% of the peers with an uptime of 2 hours will remain up for at least another 24 hours. Therefore, the only way to ensure that keywords remain available for 24 hours is to publish information about a keyword on *more than one peer*, as is done by eMule and aMule.

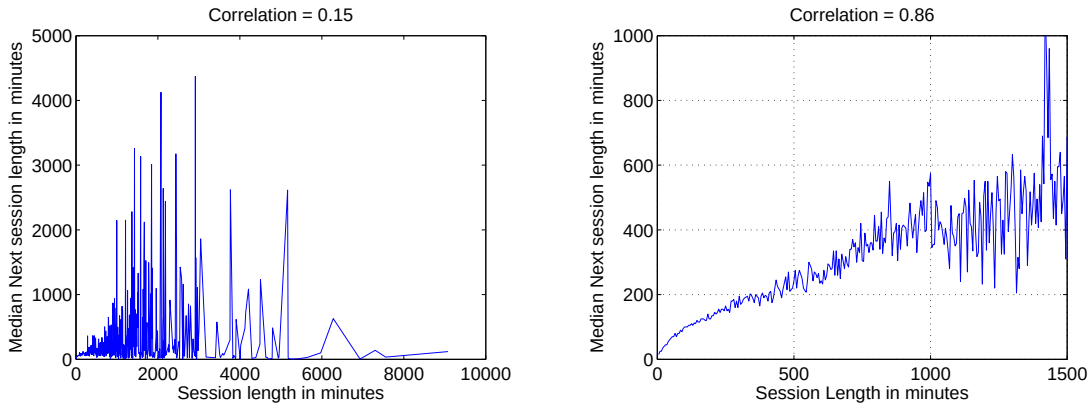
Trying to increase the lifetime of published content only makes sense for those peers that are themselves highly available. Therefore, the publication strategy needs to choose the content lifetime as function of both, the past availability of the publishing peer and the availability of the peers where the data is going to be published.

9.2.3 Next Session Time

One may ask the question whether consecutive sessions are correlated in length. If there is a strong positive correlation, one could use information about past session lengths as a predictor of the length of future sessions: If a publishing peer could

predict its session time it could then choose the optimal value for the expiration time of the information it publishes in KAD.

If we take all session length samples and compute the coefficient of correlation over consecutive session lengths we obtain a value of 0.15, which indicates that there is almost no correlation. However, if we only consider session lengths up to 1 day, there is considerable positive correlation of 0.85, which was also observed by Stutzbach (Figure 10(b) of [123]). This example nicely illustrates how incomplete data due to too short crawling duration can have a major impact on the conclusion one is able to draw from the observations. We also computed the conditional probability for session $i+1$ to last at least 1 day given that session i has lasted 1 day or longer and obtained a value of 0.34 (Figure 9.17).



(a) All session lengths.

(b) Zoom on small values up to 1 day.

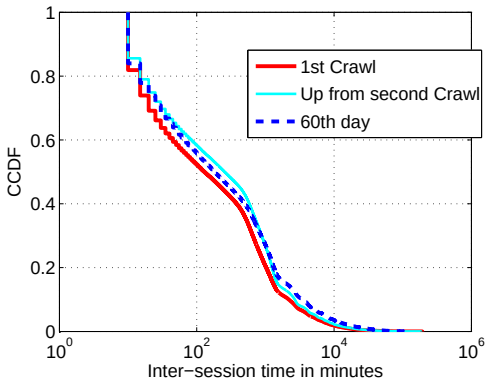
Figure 9.17: Correlation between consecutive session lengths of the peers seen.

9.2.4 Inter-Session Time

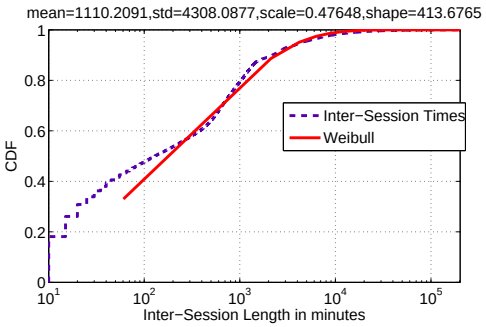
The **inter-session time** is defined as the time a peer k is continuously absent from the system, i.e., $t^j_{i+1}(k) - t^l_i(k)$ for $i \in \{1, ..., n\}$. Figure 9.18(a) depicts the CCDF of the inter-session times.

The peers seen in the first crawl have, on average, not only longer session times but also smaller inter-session times than peers seen the first time in later crawls (cf. Table 9.1). The average inter-session time is 1110 minutes for peers seen in the first crawl compared to 1349 minutes for peers seen first during crawls 2 up to 288. Peers in China have much shorter inter-session times than peers in Europe (cf. Table 9.1). The longest inter-session time observed is 177 days.

For the inter-session times we could not find a distribution that matches well our observed data. In particular, the Weibull distribution did not fit (Figure 9.18(b)).



(a) CCDF of the inter-session times of first crawl, of the first day up from the second crawl and of day 60.



(b) Weibull fit of the Inter-session time for peers of first crawl

Figure 9.18: CCDF of the Inter-Session times of the peers.

9.2.5 Lifetime of Peers

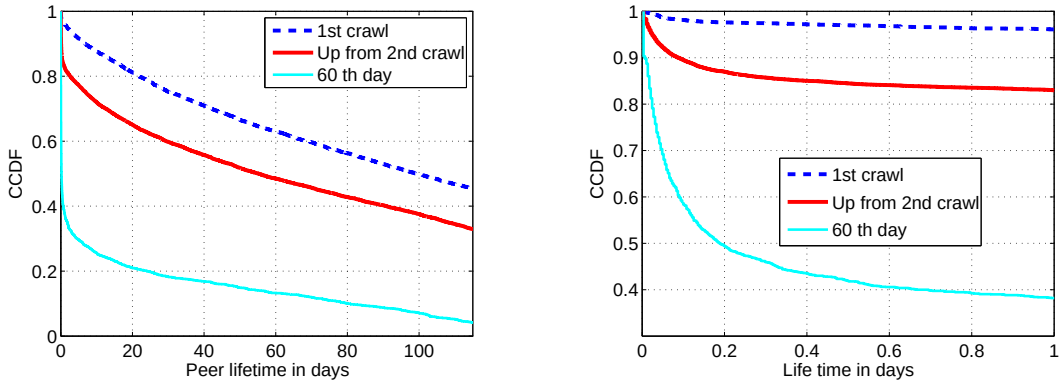
For a given KAD ID k, let $t^j(k)$ be the time this KAD ID is seen joining KAD for the first time, and let $t^l(k)$ be the time this KAD ID is seen for the *last* time. The lifetime of KAD ID k is defined as $t^l(k) - t^j(k)$. Since our crawl is of a finite duration, we can never be sure if a peer with KAD ID k will not come back after we stopped crawling. To make such an event very unlikely, we have decided to compute the lifetime only for peers with KAD IDs seen for the last time 60 days or more before the end of our crawl. We set the cut-off at 60 days, since the inter-session times seen are very rarely longer than 60 days.

Since at time $t^l(k)$ we do not know whether peer k will re-join KAD later, it is clear that our definition of lifetime gives a *lower* bound on the actual lifetime of a peer.

Figure 9.19 depicts the the CCDF of lifetime for KAD IDs seen for the first time during the first crawl, on the first day, and for the first time on the 60th day. It is striking to notice that the KAD IDs first seen on day 60 have a *much lower* lifetime than the KAD IDs seen the first day. In fact, only 40% of the KAD IDs that were first seen on day 60 will be seen for more than one day. Among the peers seen on the first day, about $2/3$

of the peers have a lifetime longer than one month and close to 45% have a lifetime longer than three months.

As we know from table 9.1, more than half of the KAD IDs first seen on day 60 are from peers in China; it is for these peers that we could clearly establish that participants change their KAD IDs.



(a) Different days (b) Zoom on the first day of lifetime

Figure 9.19: CCDF of the lifetime of peers seen during the 1st crawl, first day up from the second crawl, and on day 60.

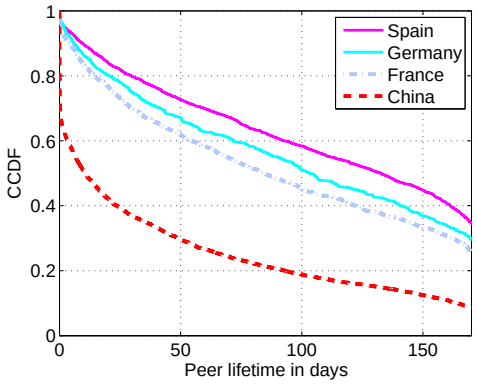


Figure 9.20: CCDF of the lifetime of the *peers* seen on the first day according to country of origin.

Figure 9.20 depicts the complementary cumulative distribution (CCDF) of the peers seen on the first day. There is a big difference in the lifetime of peers from China as compared to Europe: more than a third of the Chinese peers disappear after only one day and only 10% have a lifetime of more than 150 days, while close to 40% of the peers in Europe have a lifetime of more than 150 days.

In fact the lifetime of KAD IDs strongly depends on the number of times a peer reconnects to the system. About 30% of the Chinese peers use the same KAD ID for only one session compared to 5% for the European peers (cf. Figure 9.11(b)), which explains the difference in peer lifetime seen in Figure 9.20.

9.2.6 Lifetime of End-users

The lifetime of end-users expresses in some way the satisfaction of end-users. If users come back again and again it means that they rely heavily on peer-to-peer file sharing for accessing and exchanging content with other users.

Due to KAD ID aliasing (cf. Section 9.1.3) we can not estimate the lifetime of end-users by measuring the lifetime of KAD IDs (cf. Section 9.2.5). The only way to measure the lifetime of end-users is to analyze peers from the pivot set that we already used in section 9.1.3 to measure aliasing of KAD IDs. Since the pivot set contains over one hundred thousand peers from numerous countries we expect that these peers are representative of all peers in KAD. In Figure 9.21, the CCDF of the lifetime of end-users is plotted. End-user lifetimes are significantly larger than peer lifetimes (see Figure 9.20). 50% of the end-users have been using KAD for 6 months and more.

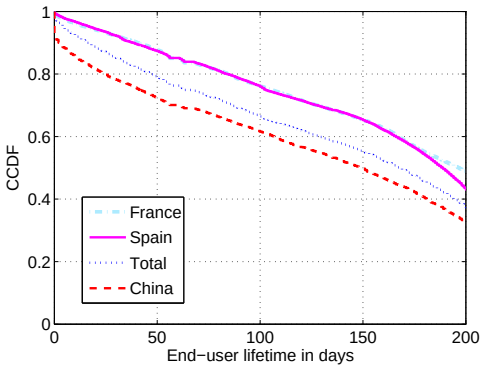


Figure 9.21: CCDF of the lifetime of the *end-users* having static IP addresses and port number.

9.2.7 Daily Availability

Characterizing availability is important for building efficient distributed applications such as overlay multicast or distributed file systems. For instance, availability guided file placement can help reduce the cost of object maintenance [79], which may potentially be prohibitive as was pointed out by Blake [9].

Availability in the case of KAD measures the *usage behavior*, i.e., how many hours a day users are connected and how they use KAD over longer time periods such as weeks.

Daily availability measures the fraction of time a peer is connected per day. Daily availability expresses the "intensity" of participation of users in the exchange of files. For a given peer P, we define **daily availability** of P as the percentage of time P was seen online that day. For a peer that was first seen at day i and last seen at day j, we will get a time series of daily availability values that has $j - i + 1$ elements. We define the **mean daily availability** as the average of those $j - i + 1$ values.

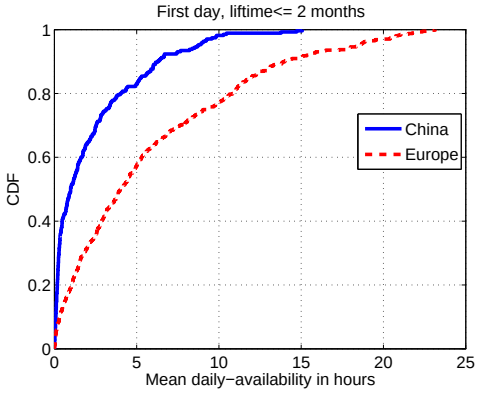


Figure 9.22: CDF of the mean daily availability of peers seen the first day.

Peers in China spend much less time per day connected than peers in Europe (Fig. 9.22). The "online times" for peers in Europe are quite impressive, with 40% of the peers being connected more than 5 hours per day and 20% even more than 10 hours per day.

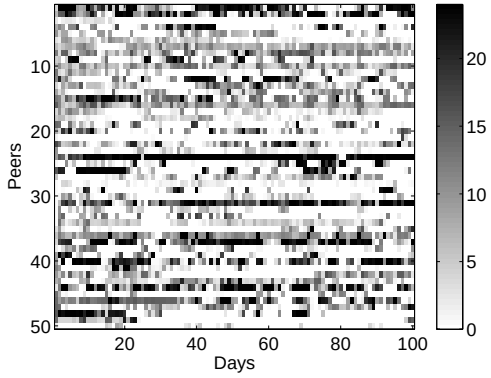


Figure 9.23: Daily availability in hours of 50 randomly chosen peers seen in the first crawl.

Figure 9.23 plots the daily availability (in hours) time series for a random set of peers over a duration of 100 days. While there are a few peers for which the daily availability changes little over time, most of the peers exhibit daily availability values that vary a lot.

9.3 Related Work

Studies measuring peer-to-peer networks may have different goals, such as analyzing the traffic patterns [102, 131]; learning about the content shared in the network [22, 41, 101]; or learning about the peers, their geographical distribution [41], their latency to the measurement site or their bandwidth [64, 101], and the user behavior expressed e.g. in session times or peer availability [8, 22, 64, 101, 123, 129]. There are also different ways to measure peer-to-peer networks, which can be either

passive or active. Passive techniques consist of (i) instrumenting a client that captures all the traffic [131] sent and received, (ii) analyzing the central log file such as the track log of BitTorrent [56], or capturing the traffic of a whole network, e.g. at the POP or an ISP [91, 102].

Active techniques consist in crawling the peer-to-peer system. In some systems, such as Napster or eDonkey, it is sufficient to contact the server(s), instead of every single peer [22, 96, 101]. If one wants to know all peers in a DHT, such as Overnet or KAD, it is necessary to contact every single peer and to query it for contacts in its routing tables. Here crawling is equivalent to a graph exploration. Finding all peers amounts to building the transitive closure of the graph. Examples of DHTs studies that employ crawling are [8, 41, 64, 96, 123].

Overnet was the first widely deployed peer-to-peer application that used a DHT, namely Kademlia. The implementation of Overnet is proprietary and its operation was discontinued in September 2006 after legal action from the media industry. Overnet has been the subject of several studies [8, 64] and up to 265,000 concurrent users have been seen online. In our study we use the active measurement approach and want to learn about peer behavior. One study relevant to our work is by Bhagwan et al. [8] where a set of 2,400 peers in Overnet was contacted every 20 minutes over two weeks. This study discusses the *IP aliasing problem* which results from the fact that many peers periodically change their IP address. Therefore, in order to properly compute session times and other peer-specific metrics, one needs to use a globally unique identifier for each peer. This study also indicates, that for systems where peers leave permanently, the mean peer availability decreases as the observation period considered increases.

KAD is the first widely deployed open-source peer-to-peer system relying on a DHT. Two other studies on KAD that are very relevant to our work have been published by Stutzbach and Rejaie. The first one explains in detail the implementation of Kademlia in eMule [122] and the second one [123] compares the behavior of peers in three different peer-to-peer systems, namely BitTorrent, Gnutella and KAD. The results obtained for KAD are based on crawling a subset of the KAD ID space. We call a continuous subset of the total KAD ID space that contains all KAD peers whose KAD IDs agree in the high order k bits a **k-bit zone**. Stutzbach and Rejaie have implemented a custom crawler that allowes them to crawl a 10-bit zone in 3-4 minutes and a 12-bit zone in approximately 1 minute [123]. A total of 4 different zones were crawled, each one being crawled for 2 days. The short duration of the crawls implies that the maximum values for some metrics such as session times or inter-session times that can be observed are naturally limited to 2 days. The paper by Stutzbach and Rejaie [123] is the most relevant to our work and we will refer to the results reported on several occasions. As we will see, our work significantly extends the findings of Stutzbach and Rejaie. It confirms some of their findings but disagrees in other places. This disagreement is in part due to the fact that a crawl duration of two days is too short with respect to some of the key metrics such as session durations.

Le Fessant at al. [41] crawled eDonkey for one week and connected 55,000 out of 230,000 peers. The geographical distribution of these peers is very similar to the one

we have observed (cf. Section 9.1.2), except for the large number of Chinese peers that we see.

In 2002, Saroiu et al. [101] presented one of the first measurement studies for Gnutella and Napster. They developed their own crawler that connects, in the case of Napster, to each of the 160 servers and asks for the connected clients. The Gnutella crawler explores the graph of neighbor relations, every crawl is stopped after two minutes, during which about 10,000 peers are discovered.

The Gnutella crawl spans 8 days and the Napster crawl spans 4 days.

For the lifetime measurements, they monitored 17,125 Gnutella peers over a period of 60 hours and 7,000 Napster peers over a period of 25 hours. They pinged each Gnutella peer every seven minutes and Napster peer every two minutes.

For both Gnutella and Napster, Saroiu et al. report median session times of about 1 hour, which is half the time compared to the peers in KAD (cf. Section 9.2.1). This difference might be due to the fact that the study of Saroiu et al. was performed 5 years ago, and that cheap broadband connections where not largely available and that large files that take a long time to download where not yet available on peer-to-peer networks.

Chu et al. [22] repeated the measurements of Saroiu et al. and extended them to a duration of six weeks measuring session lengths and content popularity. They analyzed the node availability and come to the results that due to the longer observation period, sessions with shorted duration are more dominant in the distribution than stated by Saroiu et al.

Qiao and Bustamante [96] compared the performance of structured and unstructured Overlay networks for the case of Overnet and Gnutella. For their study they performed session measurements for 7 days and reported median session times of 71 minutes for Gnutella and of 135 minutes for Overnet, which is very close to our results for KAD (cf. Section 9.2.1). In [14] the same authors present a new peer-to-peer system that makes use of the expected session times of the peers that follows a heavy-tailed Pareto distribution in order to improve resilience to churn.

Tian and Dai [129] analyzed the logs of Maze, a Chinese peer-to-peer network with about 20,000 concurrent users. All users are connected via the high speed Chinese research network. Although it is a peer-to-peer network, all peers connect every 5 minutes to a central server that writes a global log file. This enables an analysis to be made.

9.4 Design Implications

The results presented in this paper can be used to validate design choices made by the developers and improve the performance of the implementation of KAD in various ways: In today's implementation of KAD, a source key that points to the peer

that holds the content will expire 5 hours after it has been published. On the other hand, the median session length of peers is only 155 minutes and less than 40% of the sessions are longer than 5 hours (cf. figure 9.12). This means that in more than 60% of the cases, a peer that publishes a source key will leave KAD before the reference to that file expires. As a consequence, many references to sources will be *stale*, resulting in unsuccessful attempts to download that file. An improvement of the current implementation could be to first publish a source key with an expiration time much smaller than 5 hours. Each time the published source key expires, the peer that owns the file republishes the source key, progressively increasing its expiration time.

In [116] we measured that the total traffic in KAD due to *publishing* is about 100 times higher in volume (bytes) than the total *search traffic*. In a follow-up to this measurement study, we have shown how to exploit the fact that session times are Weibull distributed in order to reduce the publish traffic by one order of magnitude [18].

We have seen (cf. Section 9.2.1) that for peers with session times less than one day, the duration of the next session is highly correlated to the duration of the previous session. This means that a peer could set the initial expiration time of a source key using the value of its last session time.

Peers seen first during the very first crawl have much higher mean session durations and smaller inter-session times than peers seen later for the first time. As already suggested by Stutzbach and Rejaie [123], this fact can be exploited to find "more stable" peers without knowing anything about the history of the peers: One simply "crawls" KAD *once* and selects the peers that are online at that instant.

The design of KAD is very robust in view of non-standard behavior, such as KAD ID aliasing, which does not affect the routing and lookup performance of the DHT.

9.5 Conclusion

In this chapter we have investigated the user behavior of KAD, which is currently the only DHT deployed on a large scale. Studying KAD poses a number of unique challenges to be addressed. To obtain the necessary data,

- We have implemented a very fast and highly efficient crawler of KAD. The speed of our crawler made it possible, for the first time ever, to carry out a full crawl of the entire KAD system. It also allowed us to track the behavior of a representative subset of KAD peers (with a precision of $\pm$ 5 minutes) over a period of almost six months. To the best of our knowledge this is the longest crawl of a peer-to-peer system ever carried out.

- We need to crawl for such a long duration to unambiguously identify the peers that joined KAD for the first time and to "capture the tail" of the session and inter-session time distributions, which is in the order of months.

- To be able to cope with transient network and machine failures, we ran two crawlers in parallel and we "post-processed" our measurements to account for missing replies of peers that are overloaded.

We have carried out a full crawl once a day in order to

- Validate that crawling a single zone will return a sample of the peers in KAD that is representative of the entire KAD network.
- Obtain a subset (pivot set) of peers with static IP addresses that can be used to estimate the rate of change of KAD IDs and the lifetime of *end-users* and not only the lifetime of KAD IDs.
- Detect various "anomalies" such as thousands of peers with the same KAD ID or a company "observing" the entire search and publish traffic via thousands of *sybil* peers.

Our high resolution zone crawl lead to a number of interesting findings:

- Session times are heavy tailed following a Weibull distribution.
- KAD IDs are not necessarily persistent as was assumed up to now. Nevertheless, the most important metrics such as session times and inter-session times are not affected by the non-persistent KAD IDs.
- The total number of peers online at any time can be precisely estimated.
- Peers in China differ significantly from peers in Europe with respect to key metrics such as session time, inter-session time, peer lifetime, and daily availability.
- The majority of clients use KAD every day for many hours.
- Since most of the content is copyright protected and the sharing of such content is illegal, users take measures to reduce the risk of being tracked by changing their KAD ID frequently or by staying connected only as long as necessary to download the desired content.

The full dataset of the zone crawl and of the full crawl are available on the Web `http://www.eurecom.fr/~btroup/kadtraces`.

CHAPTER 10
Content in KAD

The KAD system is designed to prevent free-riding, anyone who retrieves a file from KAD also becomes a server for that file, and he publishes this fact to the rest of the world. Thus, new publications are a consequence of successful retrievals. To investigate the publishing process of KAD, we designed and implemented our own content spy, *Mistral*, that is described in detail in Section 8.4.

We first launched *Mistral* on the entire KAD ID space. The load on our machine was too large, and not all the queries could be satisfied and recorded. Since spying on the entire KAD ID space was not possible, in May 2007, we spied on 20 different 8-bit zones (such a zone contains the peers having the 8 first bit in common) of the KAD ID space during 24 hours[1], which allowed us to obtain a number of original results. We observed for a single 8-bit zone :

- More than 1.5 million distinct users,
- More than 1.4 million different references to published files,
- More than 42,000 different keywords were published and only 1,100 searched,
- Per minute, about 1,000 search requests, 10,000 publish requests and 25,000 route requests, which amounts to a load of approximately 400 KByte/sec for the incoming and approximately 200 KByte/sec for the outgoing traffic.
- Publishing generates ten times more messages than searching. Moreover, publish messages are ten times larger than search messages. Thus, the total number of bytes transmitted differs in *two orders of magnitude*.
- The popularity of the keywords is not at all uniformly distributed, as it was also observed for other DHTs [23, 103].
- The most popular keywords are meaningless stopwords.

[1]This is the minimum observation time required to catch also rare keywords since the typical keyword republishing interval is 24 hours.

These results led us to look into means how to reduce the publishing overhead effectively without reducing the retrieval success rate of the KAD system.

From the indexing and retrieval literature, it is well known that eliminating *stopwords* helps to reduce unnecessary searches. Stopwords are very common words of a language that do not contribute much to the power of an index; examples are "the", "a" or "what" that occur frequently without adding much meaning. When a full-text index is built, these stopwords are usually excluded. For example, in the English language, 26 stop words make up 33% of a text.

All DHT-based peer-to-peer systems index files based on the file names, and so does KAD. However, KAD's only mechanism related to stopwords is to leave out all one- and two-letter words when creating an index for a file name. This does not seem to be an efficient solution. As a consequence, we decided to

- Measure the performance of a KAD-based peer-to-peer system with *Mistral*,
- Provide a specific set of stopwords and exclude these stopwords from the index creation and search process,

All the measurement techniques Mistral makes use of are based on the **Sybil attack**, which was first defined by Douceur [33] as "the forging of multiple identities". Mounting a Sybil attack is very easy in KAD and allows to compromise the privacy of KAD users, to compromise the correct operation of the key lookup, to "delete" content, and to mount DDoS with very little resources [113]. We will relate some of our findings and point out how KAD can be used and misused.

Another type of attack, the *content pollution attack*, does not imply forging identities. To make a keyword unaccessible to the peers, a large number of fake files (sources) is published under the hash of the keyword to attack. The peers hosting the keyword are going to return these fake publications to the content search request by benign peers. We use this type of attack to disrupt the communication in the peer-to-peer network formed by the Storm bots. This network is also based on Kademlia, even if the implementation differs in some details from the KAD implementation.

In this Chapter, we will first explain the Sybil attack – and its derivatives, the Eclipse attack and the Pollution attack – since our measurements on the content are based on these attacks. In Section 10.3, we present the results obtained with our content spy Mistral. We propose to exclude stopwords from the publishing process in order to decrease the load in Section 10.4. Before concluding in Section 10.6, we present the related work.

10.1 The Sybil Attack

We want to find out in the least intrusive way what type of content is published and searched for into a zone $\mathcal{Z}$ of the KAD network. For this, one needs to introduce

*sybil*s in the zone $\mathcal{Z}$ and to make them known, so that their presence is reflected in the routing tables of the *regular*, i.e., non-sybil peers. For details, we refer to Section 8.4.

10.1.1 Eclipsing Content

A special form of sybil attack is the **eclipse** attack [105] that aims to separate a part of the peer-to-peer network from the rest. The way we perform an eclipse attack resembles very much that of the sybil attack described above, except for the KAD ID space covered being much smaller.

Again, as for the sybil attack, not the sheer number of introduced sybils is important, but, first of all, how well they are known by the benign peers. The challenge is to pollute the routing table of *every* benign peer in a way that *every* request is forwarded to a sybil. The crawling for new peers and the annoucement of the sybils therefore have to go on during the whole time of the attack. As soon as the crawling and the announcing are stopped, newly joining peers may also learn about some non-sybil peers.

In order to eclipse a particular keyword K, we position a certain number of *sybil*s closely around K, i.e., the KAD IDs of the *sybil*s are closer to the hash value of K than the KAD IDs of any real peer. We then need to announce these *sybil*s to the regular peers in order to "poison" the regular peers routing tables and to attract all the `route requests` for keyword K. Our experiments showed that as few as eight *sybil* peers are sufficient to make sure that all `search requests` for K will terminate on one of the *sybil*s. That means that the keyword is invisible, *eclipsed*, to all the participants in the KAD network. Our search attempts with a large number of clients at different positions in the hash space show that already 3 to 5 minutes after the start of the eclipse attack the routing tables of the benign peers are poisoned and there is an absolute certainty that a lookup ends on one of our *sybil*s. After stopping the attack, it takes some minutes for the benign KAD peers to recover, i.e., to remove the *sybil*s from their routing tables.

message type (messages per min)	keyword	
	the	*dreirad*
`route`	41801	818
`hello`	1091	433
`publish`	12360	290
`search`	704	49
Total incoming bandwidth (KByte/sec)	186	32

Table 10.1: Traffic seen by the *sybil*s that eclipse the keywords *the* and *dreirad*.

Note that even if the keyword K cannot be found anymore using the search algorithm employed in KAD, it does still exist on the regular peers where it was originally published.

Depending on the popularity of the content to be eclipsed, the resource consumption varies as we can see in table 10.1. This data was collected using 32 *sybil*s all running on the same physical machine. We see that it is possible to eclipse content using a very limited amount of resources.

10.1.2 DDoS Attacks

A sybil attack can also be used to launch a **DDoS** attack enlisting a large number of peers that participate in KAD. As for the previous two attacks, we need to place *sybil* peers. However, in difference to the eclipse attack where incoming search queries have been dropped by the *sybil* peer, the *sybil* peer now replies to the request and includes the IP address of the "target" to be attacked in its response.

Depending on the number of *sybil*s and their placement in zones that receive more or less search traffic, the amount of attack traffic can be controlled. We have tried such an attack against some of our own machines that were hit by an incoming traffic in the order of several Mbits/sec.

These kinds of attacks are already happening in the Internet. A news release by Prolexic from earlier this year reports [94] that DDoS attacks using peer-to-peer systems that involve more than 300,000 peers have recently been observed.

10.2 The Content Pollution Attack

In this Section, we describe the content pollution attack against the peer-to-peer network run by the *Storm Worm*. We at first give some background on botnets before we detail the attack and discuss the results.

A *bot* is a computer program installed on a compromised machine which offers a remote control mechanism to an attacker. Botnets, i.e., networks of such bots under a common control infrastructure, pose a severe threat to today's Internet: Botnets are commonly used for Distributed Denial-of-Service (DDoS) attacks, sending of spam, or other nefarious purposes [24, 52, 97]. Today, we are encountering a new generation of botnets that use peer-to-peer style communication. These botnets do not have a central server that distributes commands and are therefore not directly affected by botnet tracking. Probably the most prominent peer-to-peer bot currently spreading in the wild is known as *Peacomm*, *Nuwar*, or *Zhelatin*. Because of its devastating success, this worm received major press coverage [45, 63, 85] in which — due to the circumstances of its spreading — it was given the name *Storm Worm* (or *Storm* for

short) [119]. This malware currently is the widest-spread peer-to-peer bot observed in the wild.

For finding participating bots within the peer-to-peer network and receiving commands from its controller, the first version of Storm Worm uses OVERNET, a Kademlia-based [77] routing protocol. OVERNET is implemented by Edonkey2000, that was officially shut down in 2006, but still benign peers are online in this network, i.e., not *all* peers within OVERNET are bots per se.

In October 2007, the Storm botnet changed the communication protocol slightly. From then on, Storm does not only use OVERNET for communication, but newer versions use their own peer-to-peer network, which we choose to call the *Stormnet*. This peer-to-peer network is identical to OVERNET except for the fact that each message is XOR encrypted with a 40 byte long key.

To measure the number of peers within the whole Storm network, we ran our crawler *Blizzard* (Section 8.3) on OVERNET and *Stormnet*. The speed of our crawler allows us to discover all peers within OVERNET 20 to 40 seconds (depending on the time of day).

Using our adapted content spy *Mistral* (Section 8.4), we are able to monitor requests within the whole network.

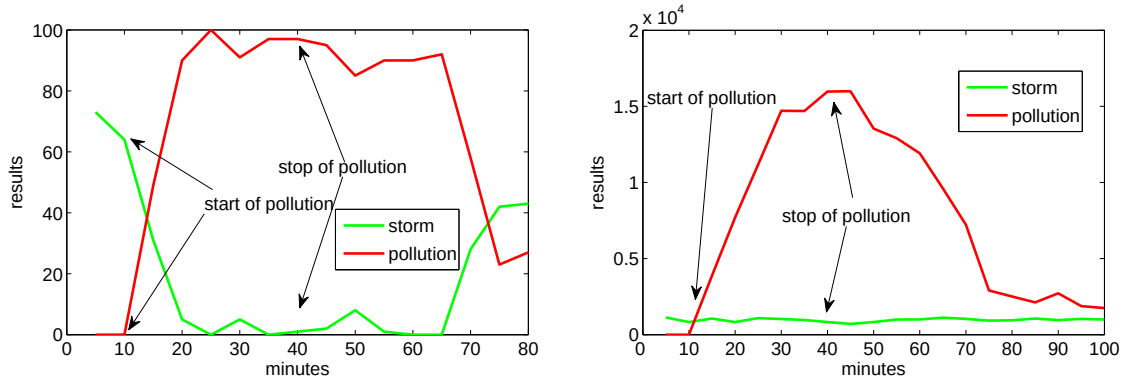
After having crawled the network and spying on the requests issued by the bots, the next step is to disrupt the communication between the bots. To prevent bots from retrieving search results for a certain key K, we publish a very large number of files using K. The goal of the pollution attack is to "overwrite" the content previously published under key K. Since the Storm bots continue to publish their content as well, this is a race between the group performing mitigation attempts and the infected machines.

To perform this attack, we again first crawl the network and then publish files to all those peers having at least the first 4 bits in common with K. This crawling and publishing is repeated during the entire attack. A publishing round takes about 5 seconds. We try to publish on about 2,200 peers during this time. About 400 of these peers accept our publications. The peers that do not respond did either previously leave the network and could not be contacted because they are behind a NAT gateway, or are overloaded and could not process our publication.

Once a search is launched by any regular client or bot, it searches on peers closely around K and will then receive so many results (our fake announcements) that it is going to stop the search very soon and not going to continue the search farther away from K. That way, publications of K that are stored on peers far away from K do not affect the effectiveness of the attack as they do for the eclipse attack.

We evaluate the effectiveness of the pollution attack by polluting a hash used by Storm and, at the same time, searching for that hash. We do this using two different machines, located at two different networks. For searching, we use `kadc` [57], an open-source OVERNET implementation, and an exhaustive search algorithm we de-

veloped. Our search method is very intrusive, it crawls the entire network and asks every peer for the specified content with key K. Figure 10.1(a) shows that the number of Storm content quickly decreases in the results obtained by the regular search algorithm, then almost completely disappears from the results some minutes after the attack is launched, and finally comes back after the attack is stopped. However, by modifying the search algorithm used, by asking all peers in the network for the content and not only the peers close to the content's hash, the storm related content can still be found (Figure 10.1). Our experiments show that by polluting all those hashes that we identified to be *storm hashes*, we can disrupt the communication of the botnet.



(a) Using the standard search.

(b) Using the exhaustive search.

Figure 10.1: The number of publications by Storm bots vs. the number of publications by our pollution attack.

10.3 Spy Results

Let us first quantify the resources required to introduce sybil peers into the entire KAD ID space using *Mistral*. Three million online peers that have to be crawled regularly with *Blizzard*, at least every two hours, to cope with the churn in the system. Each crawl accounts for 4 GByte of traffic. Afterward, the sybils must be announced to those peers. Assume that we only announce the 256 closest sybils to each peer: one announcement costs 50 bytes plus another 50 bytes for the ack; that accounts for $3,000,000 * 256 * 2 * 50$ bytes = 72 GByte. Announcements must be done periodically. On average, Sybil announcements generate about 40 MBytes/s of traffic. Moreover, the sybils will also attract search and publish messages. This clearly shows that from the bandwidth point of view it is not possible, with our resources, to place sybils in the entire network.

Placing only 256 sybils in each 8-bit zone, we get a total of 65,536 sybils. On the average, one sybil will send and receive ten 100 bytes packets per second. That generates $10 * 100 * 65,536$ bytes/s or 60 MBytes/s. Moreover, every two hours, 40 GByte are needed for the crawling of the network and the announcement of the sybils, creating

an additional 12 MBytes/s. That clearly shows that from the bandwidth point of view it is not possible to place sybils in the entire network.

We once tried to launch our spy *Mistral* on the entire KAD ID space. The load on our machine was so high that not all the queries could be satisfied and recorded. Despite all these problems, we obtained interesting results. Figure 10.2 shows that the popularity of the keywords is not at all equally distributed. The peaks related to the words "and","com","for", and "dvd" are clearly visible. All these words are meaningless stopwords. Note that this figure is incomplete, not all the popular keywords appear, since the machine running the measurements was overloaded and had to drop many requests. The load among the peers is clearly unbalanced. KAD peers that by misfortune have a KAD ID close to a peak assume much more load than other peers.

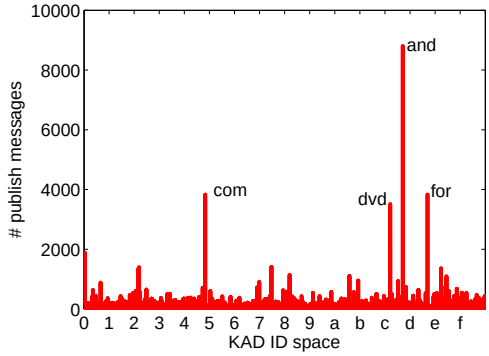


Figure 10.2: The number of publish messages in different zones of the KAD ID space.

The hash values of the search requests, the keywords publish requests, and the keywords themselves are uniformly distributed over the KAD ID space. The same is true for the source search requests and the source publish requests. We also know from our earlier measurements with *Blizzard* that the peers are uniformly distributed as well on the KAD ID space (Section 9.1.1).

This property allows us to estimate the total number S of sources (files) in the system by simply counting the number of sources in a zone. Let S_{part} be the number of sources counted in an 8–bit zone, and $\hat{S} := 256 * S_{part}$ the estimate for the total number of sources in the KAD system. Using Chernoff bounds (see [82], Chapter 4), we tightly bound the estimation error. Indeed, $Prob(|S - \hat{S}| < 45000) \geq 0.99$, which means that our estimate $\hat{S}$ most likely has an error of less than 3% for a total number of at least 80 million sources.

What type of content is behind these 80 million sources? Almost two third of it are audio files, followed by video files, software, documents, and images (Table 10.1(a)). The most frequently used audio format is mp3 with 87% of the audio files (Table 10.1(b)). Half of the files are encoded with a bitrate of 128kbps, a quarter with the higher bitrate of 192kbps. Regarding the video formats, the battle is not yet decided: 34% of avi files, 24% of wmv files, and 16% of mpg files (Table 10.1(d)). One is

not surprised that, for documents, the most frequently used format is pdf with 43% (Table 10.1(e)).

(a) The fractions of the different file types.

file type	fraction
Audio	0.61
Video	0.15
Software	0.11
Document	0.05
Image	0.05

(b) The audio formats used.

audio format	fraction
mp3	0.87
wma	0.02
mid	0.01

(c) The audio bitrates (kbps) used.

bitrate (kbps)	fraction
128	0.50
192	0.26
160	0.06
320	0.05
256	0.02

(d) The video formats used.

video format	fraction
avi	0.34
wmv	0.24
mpg	0.16
mkv	0.09
rmvb	0.04
asf	0.04
mpeg	0.03
rm	0.02

(e) The document formats used.

document format	fraction
pdf	0.43
txt	0.17
doc	0.10
nfo	0.06
htm	0.02
html	0.01
ppt	0.01

Table 10.2: An overview of the file types and the file formates used in KAD. If the fractions do not sum up to 1.00, marginal types/formats have been omitted.

The file sizes strongly depend on the type of the files. For audio, about 90% of the files have a size between 2MB and 10MB. Whereas for video the distribution is larger, the sizes of a typical divx movie (about 730MB), and of a movie encoded with half the tv resolution (360MB) are noticeable. More than half of the documents are less than 10KB long, these are probably readme files or the like (Figure 10.3).

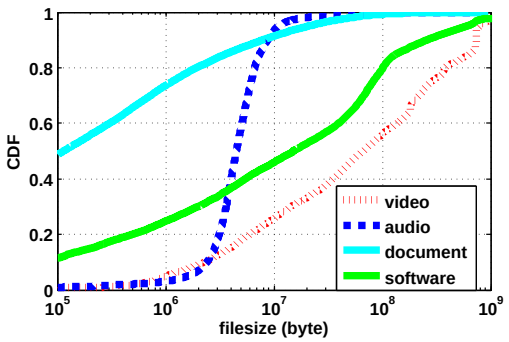


Figure 10.3: The CDF of the file sizes split by file type.

We did also encounter *pollution*, what we define as files that are announced with a wrong file name. For each file, a hash on its content is computed, therefore files can be securely distinguished from each other. Mainly porn movies are announced with file names of recent cinema movies. A file that has two (or more) different names is not necessarily polluted, a user may simply have removed some words (e.g. divx, fr) from the title. We consider about 5% of the files to be polluted (Figure 10.4). The maximum number of different titles for one file was 395.

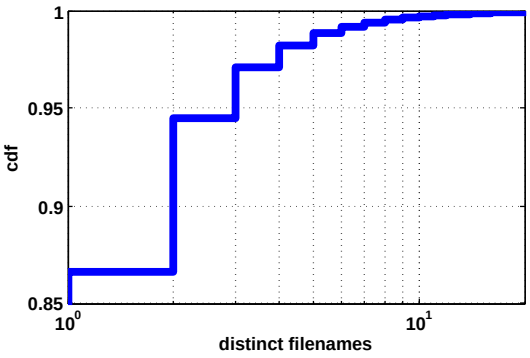


Figure 10.4: The number of distinct file names per file (identified by the hash of its content).

10.4 Reducing the Publish Actions

Our measurements show that there are ten times more publish messages than search messages. This is due to the fact that search actions have to be induced by a human participant in contrast to publish messages: the client application publishes all the shared content regularly. Moreover, a publish message is 10 times bigger than a search message since it does not only contain a keyword but also metadata describing the published content. Thus we focus on improving the performance of KAD by reducing the number of publish actions.

The number of times a keyword publication is observed versus the ranking of the keyword for the 8-bit zones `0xe3` and `0x8e` are shown in Figure 10.5 in log-log scale. Rank 1 is the most popular keyword. If each curve was to be a straight line, then the popularity of keywords would follow a Zipf-like distribution (i.e., the probability of seeing a publication message for the i'th most popular keyword is proportional to $1/i^{\alpha}$ [107]). We used Matlab's curve-fitting tools to estimate the value of α for the curve. The value of α is the same for all zones: $\alpha \approx -1.63$. It is near the most popular keywords where the zones differ by an order of magnitude.

We picked two zones as examples. The zone `0xe3` contains the keyword "the", whereas the zone `0x8e` does not contain any popular keywords. The keyword "the" in zone `0xe3` accounts for 30% of the total load in the zone. In total 1,518,717 publish requests with the keyword "the" hit our sybils in 24 hours. Whereas, in zone `0x8e`, the most popular keyword accounts only for 5% of the load. In this zone, the most popular keywords are almost equally popular.

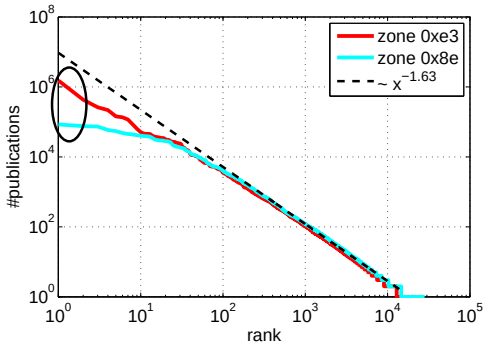


Figure 10.5: The number of publications per keyword for two different zones.

Not only the zone as a whole benefits from not publishing stopwords, but especially single peers close to the stopwords' hashes. Qiao and Bustamante [96] nicely describe the load balancing mechanism of Overnet, which relies on the same mechanisms as KAD. They state that peers close to hot spots are experiencing only 50% more overhead than other peers. However, all their measurements were done with search requests only. They simply show how widely a keyword is spread over the hash space. However, our measurements with *Mistral* show that the publish messages always hit those machines first that are close to the hash of the popular keyword. Only if these machines are overloaded, the publisher tries to contact machines more distant from the hash of the keyword. Figure 10.6 shows the number of queries that hit our ten most loaded sybils in the two zones `0xe3` and `0x8e`. The popular keyword "the" in zone `0xe3` is mainly responsible for the high load on these syblis. The sybils with a lower rank have the same load in both zones.

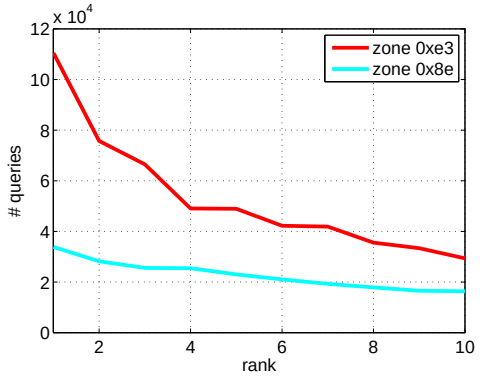


Figure 10.6: The number of queries received by the sybils for two different zones.

In figure 10.7, the returned load values that are obtained by publishing to keywords whose hashes are very close to each other in the hash space are plotted: "the" and "cassaforte". Close to the hash value of "the", the peers are overloaded in such a way that also the keyword "cassaforte" can not be published.

The popular keywords that make the difference between the zones are all stopwords. Table 10.3(a) shows specific stopwords for KAD file names which augment the set believed to be used in Google (Table 10.3(b)). We propose to use the union of Ta-

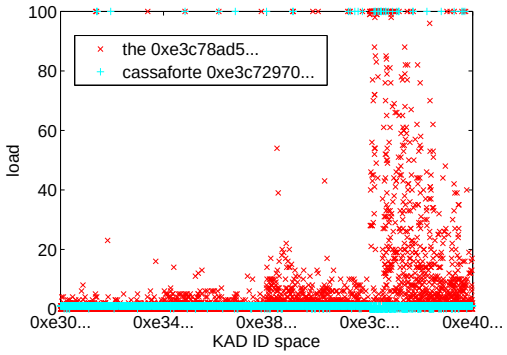


Figure 10.7: The number of queries received by the sybils around two keywords.

ble 10.3(a) and Table 10.3(b) as stopwords. The number of peers on which a stopword is published as well as the number of files containing the stopword have been determined by first crawling the peers around the stopword with *Blizzard* and then by querying all those peers for the stopword. Excluding these words from the publishing process does not affect the usability of the system, as a user can still specify the filetype (documents, music, movies, etc.) he is looking for.

A DHT is not meant to be browsable, it is not designed to keep a complete list of the content. However, for the moment, this is exactly what happens when publishing keywords like “avi”, “english”, “french”, “divx”, “dvdrip”, etc. Users who want to download an English movie simply search for “dvdrip” and “english”. These search requests result in a random selection of all files containing these keywords in their title.

Not only the popularity expressed by the number of publish request, but also by the number of distinct publishers, identified by their IP addresses, follows a Zipf distribution.

Unlike for the keywords, the popularity of the sources is much better balanced inside the zones and between the zones. The most popular source accounts only for 0.1% of the source publish traffic.

Improve the content availability Changing the implementation of KAD in a way that the content is not published every 24 hours on 10 peers but every 5 days on 50 peers, which results in the same amount of overhead, one can significantly improve the availability of the content (Figure 10.8).

Trying to augment the live time of published content is only important for those peers that are highly available themselves. It is therefore necessary to know its own past availability as well. If a peer is very volatile, it does not make sense to publish metadata pointing to itself on high available peers, since these pointers will be broken if the peer itself is going off line. That means that the publish strategy has to be adopted to the peer’s own past availability as well as to the availability of the peers where the meta-data is going to be published. This observation leads us to design an

(a) The stopwords for KAD

stopword	# peers	# files
avi	491	8101
xvid	479	13683
192kbps	437	8005
dvdscreener	413	12343
screener	433	7377
jpg	456	10529
pro	303	8378
mp3	482	12019
ac3	424	8045
video	468	10478
music	335	8558
rmvb	454	13643
dvd	450	10194
dvdrip	560	13235
english	388	7849
french	377	9468
dreirad	28	30

(b) The Google stopwords

stopword	# peers	# files
about	513	7608
are	330	7282
com	463	11550
for	549	12303
from	399	8345
how	542	8282
that	423	9148
the	487	14502
this	452	8510
what	394	7710
when	294	7241
where	431	9445
who	302	7742
will	458	7976
with	338	8543
www	391	11203
and	577	13706

Table 10.3: The Google and KAD stopwords with more than two letters, the number of peers storing them, and the number of files containing them. For comparison, the rare keyword "dreirad" is shown.

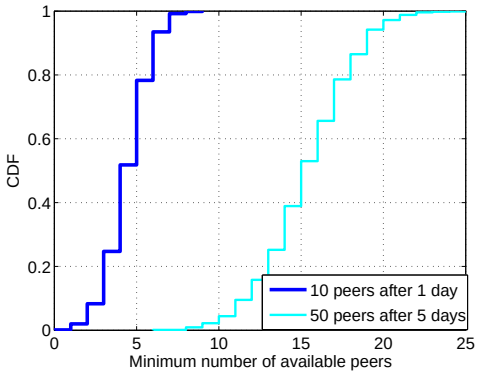


Figure 10.8: The minimum availability of 10 publishes during 1 day compared to 50 publishes during 5 days.

improved publish scheme that maintains the same degree of availability for the information published while reducing the amount of traffic by one order of magnitude [18].

10.5 Related Work

Attacks in KAD There has been a small body of work that addresses the issue of DDoS attacks using peer-to-peer systems. Naoumov et al. [84] discuss attacks for the case of the now defunct Overnet system. Since routing in Overnet resembles closely routing in KAD, their findings are very relevant to KAD. Two types of attacks are identified: **Index poisoning** attacks where bogus records are inserted into the overlay in order to direct peers searching for content to a target host that will become the victim of the DDoS attack. **Routing poisoning** attacks where many peers are tricked into adding the target host into their routing table. As a consequence the target host will receive a lot of signaling (query, publish and maintenance) traffic.

El-Defrawy et al. [36] have investigated index poisoning attacks in BitTorrent and Athanasopoulos et al. [4] discuss how to launch DDoS attacks in Guntella, an unstructured peer-to-peer file sharing system.

There exist quite a few proposals in the literature to improve the security of DHTs.

DHT-based overlay systems are susceptible to various attacks launched by malicious peers that may corrupt data, deny response to lookup queries, or impersonate other peers so that data objects may be stored on rogue peers.

In DHTs-based systems, each node has a global identifier ID, which is generated when the client application is started for the first time. If an attacker controls a fraction, even small, of nodes with smartly chosen IDs, it can *eclipse* correct nodes and prevent correct overly operation. The malicious nodes may be different entities or the same entity with many identities (IDs).

Sit et al. [106] provide a clear description of security considerations that involve peers that do not follow the protocol correctly: routing deficiencies due to corrupted routing lookup nd updates; vulnerability to partitioning when new peer joins and contacts malicious peers; lookup and storage attacks; inconsistent behaviors of peers; denial of service attacks; and unsolicited responses to a lookup query. They argue that the peer's identifier assignment must be done in a verifiable way, and that the identifier must not be chosen by the node itself. However, they mention that a central identification authority is not desirable in all situations.

Douceur [33] was the first to consider the problem of multiple identities in the context of DHT-based peer-to-peer systems (The Sybil attack). He showed that without the use of a centralized authority that certifies all nodes, it is impossible to prevent this attack.

Castro et al. [20] presented a design and analysis of techniques for secure peer joining. They propose to certify the node IDs by a set of trusted certification authorities (CAs). Node ID certificates are signed by the CAs, which use a public key that must be known by all network nodes. To prevent an attacker from obtaining certificates, they propose to bind the ID to peer's IP address, or require paying money for certificate.

Rowaihy et al. [99] propose an admission control system that mitigates Sybil attacks by adaptively constructing a hierarchy of cooperative admission control nodes. This creates a tree structure with static root. A node wishing to join the network is serially challenged using a hash puzzle by the nodes from the leaf to the root. Each challenger node creates a cryptographic puzzle based on a hash function and the solver has to invert the hash. As hash-functions are non-invertible, the solver must use brute force to find the solution, which will require a large number of attempts. This solution relies on the limitation of computational power of the joining node, however, it may still allow a resourceful attacker to launch a substantial attack, especially if the potential for damage is disproportionate to the fraction of the system that is compromised.

Yu et al. [139] propose SybilGuard, a protocol for limiting the corruptive influence of the Sybil attack. SybilGuard is based on social network among user identities, when an edge between two identities indicates a human-establish trust relationship. Malicious users can create many identities but will have only few trust relationships. The deployment of SybilGuard requires the existence of a well-connected social network, which not the case of todays DHT-based peer-to-peer systems.

While a successful Sybil attack can be used to mount an Eclipse attack, Eclipse attacks are possible even in the presence of an effective defense against Sybil attacks. To defend against eclipse attacks, Castro et al. [20] proposed the use of Constrained Routing Tables (CRT), where a node's neighbor set contains nodes with identifiers closest to well-defined points in the identifier space, which leaves no flexibility in neighbor selection and therefore prevents optimizations like proximity neighbor selection, an important and widely used technique to improve overlay efficiency [21, 47]. In addition to CRT, Singh et al. [104, 105] propose to bound the in- and out-degree of overlay nodes, and present a defense strategy based on anonymous auditing of nodes' neighbor sets. If a node has significantly more links than the average, it might be a malicious node, and then it can be removed from the neighbor sets of the correct nodes.

Stopwords in peer-to-peer systems Stopwords have been used for decades in indexing and retrieval; we thus concentrate on the use of stopwords in the context of peer-to-peer systems.

In [92] and [61], the authors describe an indexing system for the World Wide Web based on a peer-to-peer system. Their idea is that peer-to-peer systems should be able to build good search indexes in a distributed fashion, enabling Web searches in a much more scalable way than traditional (centrally coordinated distributed) search engines. These papers introduce the concept of "Highly Discriminative Keys" (HDKs): these are a selection of "rare" keys occurring in a text document. Based on these HDKs, they build a peer-to-peer index that they evaluate experimentally afterwards. They use up to 120,000 Reuters news articles; and then apply 250 English stopwords and a stemmer when they construct a search term from the full text. This is different from our approach: we do not intend to support full-text searches over

the entire document, we are just trying to enhance the indexing for file names in peer-to-peer systems.

The authors of [2] present a general four-layer architecture for a peer-to-peer-based information retrieval system that is used to build a scalable index. They illustrate the functionality of their approach by describing how a concrete indexing system could be built with those four layers; in this context, stopwords are used to improve the precision on layer 4. Again, this architecture is more general and aims at building a full-text search engine. Their system is not implemented and no performance measurements are presented in the paper.

In [124], another approach for scalable Web search is proposed. The authors use a technique developed by R. Fagin to merge the result sets of single-term queries more efficiently than by just loading all of them onto a single site. Fagin's idea is based on sorted inverted lists that are efficiently merged across sites. The authors exclude stopwords in their searches, without specifying details. Experimental results with 120 million Web pages show a low communication overhead for multi-site queries based on Fagin's idea. However, it is not quite clear where peer-to-peer technology is used, and again, unlike in our system, the purpose is the construction of an efficient full-text index.

Detailed measurement results from a study of Gnutella and Overnet (a precursor of KAD) are presented in the paper of Qiao and Bustamante [96]. Among other things, the authors evaluate the performance of queries in Overnet. Of particular interest to our work are their results on queries to popular keywords (stopwords are very popular keywords). They conclude that these are handled well by Overnet, because it distributes the query load to multiple peers whose hash IDs are "close enough" to the hash of the keyword: the more popular the keyword, the broader the hash range. Our measurements contradicted this conclusion: first, considerable overhead is generated by initially querying the peer with the closest hash to the popular keyword who answers with "too busy"; this goes on with an gradually less precise hash value until a peer is found that is able to answer. Thus, a considerable additional load is imposed on the peers next to popular keywords. Second, we do not only consider the querying, but also the publishing load, which is much higher.

10.6 Conclusion

Distributed systems for content sharing are presumably believed to be more robust against attacks than centralized systems having a single point of failure. However, in practice, this may not be the case as long as the Sybil attack is possible. We have discussed the implications of the Sybil attack in the case of KAD, which is the largest DHT currently deployed:
The privacy of the end-users can easily be compromised, KAD itself can be arbitrary disrupted, and the peers that participate in KAD can be enlisted against their will to participate in a DDoS attack. Any of these attacks can be launched from a single PC connected to the Internet via a broadband connection. Another type of attack we

studied is the pollution attack. We presented experimental results of the feasibility of this kind of attack against the Storm worm.

We have reported our findings obtained from spying on KAD, the largest currently deployed DHT. We developed *Mistral*, a content spy, that allows us to gain an overview of the content published and searched in KAD. Our observations show that the publication process in KAD is responsible for more than 90% of the total network traffic. Moreover, we note that the load is highly unbalanced between the peers. The peaks of load are due to very popular keywords that are most often meaningless stopwords. We then have proposed to add a stopword exclusion step into all KAD based peer-to-peer systems. Our results show how this equalizes the load on the peers storing the keywords, and, as a consequence, improves the overall system performance. There is no drawback to stopword exclusion since stopwords do not carry much meaning.

CHAPTER 11
Content Access in aMule

The previous Chapters describe Kademlia in a general way, followed by measurement results from the eDonkey network. In this Chapter we analyze in detail the Content Access in the aMule implementation of KAD. Note that for other implementations of KAD the details may differ.

The aim of our study [115] is to evaluate the performance of the current implementation of content management. We identify its basic building blocks and we analyze the interactions among them.

The main contribution of our work can be summarized as follows:

- We develop a qualitative analysis of the current implementation to understand the impact of the design parameters on the latency of the overall content publishing/retrieval process;
- We characterize through measurements many interesting properties of the KAD P2P system, such as the probability that an entry in the routing table is stale, or the round trip delay of the messages;
- We evaluate through measurements the performances, in terms of overall content retrieval latency, the number of hops needed, and message overhead, of the content retrieval process;
- We propose an alternative approach – called *Integrated Content Lookup* – for the content retrieval process, by strongly coupling it with the lookup process and we develop a qualitative analysis of this scheme.

The analysis highlights some performance issues with the current implementation: the decoupling of the lookup phase and the content retrieval phase has an adversarial impact on the performance of the retrieval process.

These issues are addressed by the Integrated Content Lookup scheme we propose. The measurement-based characterization of the KAD P2P system shows that (i) a large fraction of peers in the routing table that are stale and (ii) the empirical distri-

bution of the message delay presents a non-negligible tail. These results should be taken into account in the design of the content management process, since they have a strong impact on the overall lookup delay.

11.1 Architecture

The basic operations that each node has to perform can be grouped into two sets: routing management and content management. Figure 11.1 shows some of the basic building blocks of the software architecture.

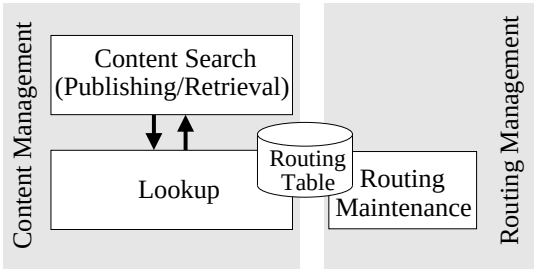


Figure 11.1: Software architecture of KAD.

Routing management takes care of populating the routing table and maintaining it. The maintenance requires to update the entries – called **contacts** – and to rearrange the contacts accordingly. A peer stores only a few contacts of peers that are far away in the KAD ID space and increasingly more contacts to peers closer in the KAD ID space. If a contact refers to a peer that is offline, we define the contact as **stale**. In order to face the problem of stale contacts due to churn (departure of peers), KAD uses redundancy, i.e., the routing table stores more than one contact for a given distance. The routing management is responsible also for replying to route requests sent by other nodes during the lookup (Sect. 11.1.1). Since in this chapter we focus on content management, we do not go into the details of the routing procedure (the interested reader is referred to [122]). The only information we use is the probability that a contact contained in the routing table is stale: p_{stale}.

Content management takes care of publishing the information about the objects the peer has, as well as retrieving the objects the peer is looking for. We summarize these two operations with the term **content search**, since they actually use the same procedure (Sect. 11.1.2). In both cases the peer has a *target* KAD ID (of the objects it wants to publish or it wants to retrieve) that it needs to reach. The KAD ID of an object is obtained by hashing the keywords in its filename. Since the peer routing table does not contain the KAD ID of all peers, the peer needs to build a temporary contact list, called **candidate list** or simply *candidates*, which contains the contacts that are closer to the target. The temporary list building process – called **lookup** – is done iteratively with the help of the other peers. Since the lookup process and the content search process represent the focus of our study, we explain them in detail in the following sections.

Note that the content management operations deal also with reply to content searches made by other peers. This aspect (not shown in Fig. 11.1) is not considered in our study since it is not relevant for us.

11.1.1 Lookup

The lookup procedure is responsible for building the candidate list with contacts that are closest to the target KAD ID, i.e., contacts with the longest common prefix to the target. The procedure, along with the main data structures, is summarized in Procedure `Lookup`.

Procedure `Lookup`

Data: $hash(128bit)$: target = hash of the target
Data: *list*: candidates = peers to query, ordered by their distance to target
Data: *list*: queried = peers queried with route requests
Data: *list*: answered = peers that replied to route requests
Data: final *int*: α = initial degree of parallelism
Data: final *int*: β = number of contacts requested
Data: final *int*: t = seconds to wait for route request messages
Data: *timestamp*: timeout `/* timeout for route request messages */`
Data: *int*: lookuptime = 0 `/* time the lookup process is running */`
Data: *int*: objectcount = 0 `/* number of object references received */`
Data: *int*: contentreplies = 0 `/* number of content replies received */`

```
1  Initialization:
2      candidates.insert(50 closest peers to target from our routing table);
3      send route request(target,β) to first α candidates;
4      queried.insert(first α candidates);
5      timeout = now + t;
6
7  When route response is received do
8      timeout = now + t;
9      answered.insert(sender);
10     foreach contact ∈ response do
11         if contact not ∈ candidates and contact not ∈ queried then
12             candidates.insert(contact);
13             if dist(contact,target) < dist(sender,target) then
                   /* approaching the target */
14                 if contact is in the α closest contacts to the target then
15                     send route request(target,β) to contact;
16                     queried.insert(contact);
17
```

The source peer first retrieves from its routing table the 50 closest contacts to the destination and stores them in the candidate list. The contacts are ordered by their

distance to the target, the closest first. Most probably those 50 contacts are too far away from the target, so that the desired content is not stored on them.

The discover process is done starting from this initial candidate list in an *iterative way*. The source peer sends a request to the first α contacts (by default $\alpha = 3$). The request is called *route request*. The source peer remembers the contacts to which a route request was sent. A route request asks by default for $\beta = 2$ closer contacts contained in the routing tables of the queried peers. A timeout is associated to the lookup process. In case the source peer does not receive any reply, it can remove the stale contacts from the candidate list and it can send out new route requests.

As soon as one route response arrives, the timeout is reset and for each of the β contacts in the response it is checked if the contact has not already been queried and it is not already in the candidate list. A route request is sent if (i) the new contact is closer to the target than the peer that provided that contact, and (ii) it is among the α closest contacts to the target. This implies that in the extreme case for every of the α incoming route responses $\min(\alpha, \beta)$ new route requests are sent out.

If the returned contact is not among the α closest known contacts it is simply stored in the candidate list: for later use in case some queried nodes do not respond to the route requests.

Figure 11.2 illustrates an example of the lookup process. On the top we show the evolution of the candidate list, where we use the flags 's' and 'r' to record if a route request has been sent or a route response has been received respectively. α is set to 3 and β is set to 2. The initial list is composed of contacts a, b, c and d. The distance in the vertical axes indicates the XOR-distance to the target. At the beginning, the source peer sends a route request to the top α contacts a, b and c. Contact c is stale and will never reply. The first response comes from b and contains β contacts, e and f, that the source peer does not know. The new contacts are inserted in the candidate list: since they are closer to the target than the other candidates, a route request is sent to them. At this point the response of a arrives. The new contacts, g and h, are inserted in the candidate list. Since contact h is not among the top α contacts, no route request is sent to h. After some time, the source peer receives the response of e, but only one of the contacts is inserted in the candidate list, since the other one is already present in the list.

The Procedure `Lookup` terminates when the route responses contain only contacts that are either already present in the candidate list or farther away from the target than the other top α candidates. At this point, no new route request is sent and the list becomes *stable*. The stabilization of the candidate list represents a key point for KAD. In fact, the source peer has to exhaustively search for all the contacts around the target. We show in Sect. 11.2 how the stabilization influences the performance.

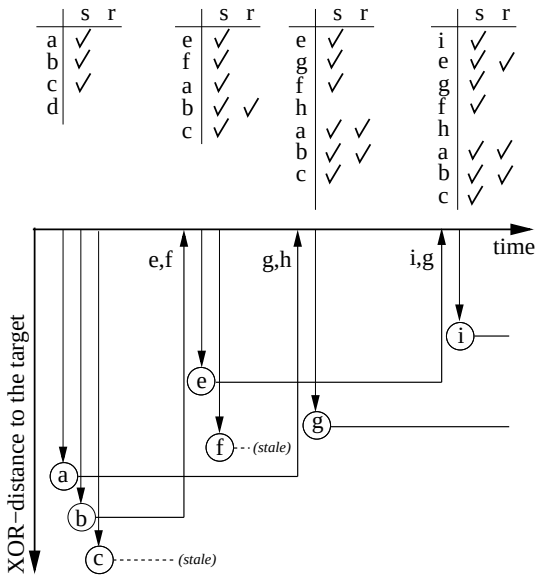


Figure 11.2: Example of lookup ($\alpha = 3$; $\beta = 2$).

11.1.2 Content Search

When the candidate list becomes stable, the source peer can start the content search process. The designers of KAD decided to consider a contact *sufficiently* close to the target if it shares with it at least the first 8 bits. The space of KAD IDs that satisfy this constraint is called **tolerance zone**.

Each candidate that falls in the tolerance zone can be considered for storing or retrieving a reference. The process is described in Procedure `Content Search`.

In the implementation of KAD, there is no direct communication between the Procedure `Lookup` and the Procedure `Content Search`, i.e., when the candidate list becomes stable, the Procedure `Lookup` does not trigger the Procedure `Content Search`. The stabilization of the candidate list means that in the last t seconds no route responses are received, where t is the *timeout* set to 3 seconds by default. This can happen for two reasons: the closest peers to the target have been found or the queried peers did not reply, i.e., the top α contacts in the candidate list are stale, or overloaded, or the messages were lost.

The solution adopted by KAD to handle these two different situations is a periodic execution of the Procedure `Content Search`: every second the procedure checks if the candidate list has been stable for at least t seconds. In this case, the procedure iterates through the candidate list: a content request is sent if (i) a route request was sent to the contact, (ii) the contact replied with a route response and (iii) the contact belongs to the tolerance zone. The content request contains a '`store reference`' type in case of publishing and a '`search reference`' type in case of content retrieval (line 13). When the procedure iterates through the candidate list and finds a contact that has not been queried, it sends a (single) route request, actually restarting the Procedure `Lookup`. This is useful in case the lookup gets stuck (line 15).

Procedure `Content Search` (Publish or Retrieval)

```
1  Every 1 sec do
2      if not stopsearch then
3          lookuptime++;
4          if lookuptime > 25 then
5              stopsearch ← true;
6          if candidates is empty then
7              stopsearch ← true;
8          if timeout ≤ now then
               /* candidate list is considered stable                    */
9              for i ← 0 to candidates.size do
10                 contact = candidates.get(i);
11                 if contact ∈ queried then
12                     if contact ∈ answered and contact in tolerance zone around target then
                           /* the timeout triggers the actual content search.
                                                                          */
13                         send content request(TYPE, target) to contact;
14                         candidate.remove(contact);
15                 else
                       /* the timeout triggers a new route request.      */
16                     send route request(target,β) to contact;
17                     queried.insert(contact);
18                     return;
19
20 When content response is received do
21     contentreplies++;
22     if peer is publishing then
23         if contentreplies > 10 then
24             stopsearch ← true;
25             return;
26     else
27         foreach object ∈ response do
28             objectcount++;
29         if objectcount > 300 then
30             stopsearch ← true;
31
```

When a content response is received, a counter is incremented. In case of content publishing, the maximum value for this counter is set to 10: in order to face churn each reference is published to 10 different peers that belong to the tolerance zone. In case of content retrieval, the response contains one or more objects with the requested reference and the maximum value for the counter is set to 300, i.e., at maximum 300 objects that contain the reference are accepted.

The main loop is stopped for one of the following three reasons: either the maximum search time is reached (lines 4-5), or there are no more contacts to query (lines 6-7), or enough content response have been received (lines 22-30).

11.2 Analysis of the Content Search Process

The content management process in KAD is divided into two procedures – `Lookup` and `Content Search`. The latter contains in a single module both content publishing and retrieval. Nevertheless, the aims of the two tasks – publishing and retrieval – are completely different. On the one hand a peer should try to publish the different replicas as close as possible to the target: this requires a candidate list to be stable, a result that can be obtained with large timeouts – note that, as explained above, a stable candidate list does not necessarily mean that the contacts are close to the target. On the other hand, a peer should look for the content as soon as it is sufficiently close to the target, i.e., when it enters in the tolerance zone: in this case a stable candidate list is not necessary.

In this section we analyze the impact on the performance of the content management approach adopted by KAD. The main performance metric for the content search process is the *overall lookup latency*, i.e., how long it takes to reach the target and find the content. The delay is mainly influenced by the following parameters:

p_{stale}	probability that a contact is stale;
d	round trip delay between two peers;
h	number of iterations (*hops*) necessary to reach the target;
α	number of route requests sent initially;
β	number of closer contacts asked for by a route request;
t	time waited for route response messages.

While p_{stale}, d and h cannot be controlled by the content management process, α, β and t do depend on the implementation.

11.2.1 Qualitative Analysis of the Latency

Lookup Latency.

For the analysis of the delay, let $F_{\text{RTT}}(d)$ be the cumulative distribution function (CDF) of the round trip delay for the single hop (see for instance the empirical CDF found with measurements and shown in Figure 11.5). At the first iteration (hop and iteration are used interchangeably) α messages are sent. We assume that the probability that all the α contacts are stale, $p^{\alpha}_{\text{stale}}$, is negligible.

Among the initial α messages, only $\alpha(1-p_{\text{stale}})$ replies are received. At each response, $\gamma = \min(\alpha, \beta)$ messages can be possibly sent out. The maximum number of messages at the second hop, $\rho_{2,\,\text{max}}$, is then $\alpha(1-p_{\text{stale}})\gamma$. In the following hop, only a fraction of $(1-p_{\text{stale}})$ of contacts reply and each response can trigger γ new requests. In general, the maximum number of messages at hop i, $\rho_{\text{i, max}}$, is

$$\rho_{\text{i, max}} = \alpha[(1-p_{\text{stale}})\gamma]^i \tag{11.1}$$

and the cumulative maximum number of messages up to hop h, $\rho_{\text{h, max}}$, is

$$\rho_{\text{h, max}} = \alpha \sum_{i=0}^{h-1} [(1-p_{\text{stale}})\gamma]^i. \tag{11.2}$$

In practice, some contacts in the replies are already known or they are not inserted in the top α positions of the candidate list, so the actual total number of messages sent up to hop h will be $\overline{\rho}_h \leq \rho_{\text{h, max}}$. Figure 11.3 shows $\rho_{\text{h, max}}$ and $\overline{\rho}_h$ for two settings for the parameters α and β. The value of p_{stale} used to compute $\rho_{\text{h, max}}$ and the value of $\overline{\rho}_h$ have been found by measurements as will be explained in Sect. 11.3. We consider up to three hops, since, as we will see in Sect. 11.3, more than 90% of the lookups reach the target in less than four hops. The actual number of messages sent is close to the maximum we computed in case of default values for α and β (3 and 2 respectively). If we increase α and β both to 4, we receive more duplicates or not interesting contacts, thus the actual total number of messages is far less than the maximum.

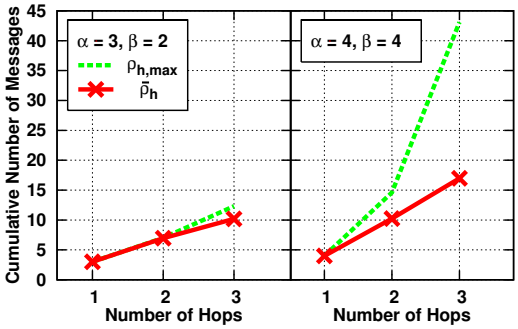


Figure 11.3: Number of messages sent during lookup.

The candidate list stabilizes only *after the last response* is received, thus the stabilization time corresponds to the maximum round trip delay over all the route requests that were sent. To simplify of the analysis, we assume that all messages are sent at the beginning[1]. The CDF of the lookup delay can be found considering that the maximum of two random variables, which corresponds to the product of their CDFs (see [19], Eq. 2.6), thus we obtain

$$F_{\text{lookup}}(d) = F_{\text{RTT}}(d)^{\overline{\rho}_h}\ . \tag{11.3}$$

[1]This is an unrealistic assumption that provide optimistic results; for our purpose, this coarse analysis is sufficient to understand the impact of the parameters.

If we increase α or β (or both), $\overline{\rho}_h$ increases, i.e., more messages are sent. The higher $\overline{\rho}_h$ is, the longer the source peer has to wait for the candidate list stabilization, since it has to wait for all the responses. This is the contrary of what one would expect, namely that sending more messages should increase the chances to reach the target faster. This means that with the current scheme it is not possible to reduce the lookup latency by increasing the parameters α and β.

Content Retrieval Latency.

Once the candidate list is stable, the lookup process terminates. At this point the content retrieval process waits for t seconds (timeout) before starting to send the content requests. This adds to the overall latency t seconds, plus a random delay uniformly distributed between zero and one second, due to the periodic execution of the Procedure `Content Search`. Moreover there is an additional round trip delay due to the content request message.

Overall Lookup Latency.

In summary, the overall latency of the content retrieval process is composed by different terms. Let $f_{\text{lookup}}(d)$ be the probability density function (PDF) of the lookup latency, i.e., the derivative of $F_{\text{lookup}}(d)$ found in Eq. (11.3). The PDF of the overall lookup delay, $f_{\text{overall}}(d)$ can be found by considering that the sum of two random variables corresponds to the convolution of their PDFs, denoted with the symbol '$*$'. We obtain

$$f_{\text{overall}}(d) = f_{\text{lookup}} * \delta_t * \text{Unif}_{(0,1)}(d) \tag{11.4}$$

where δ_t is the Dirac's delta function translated in t (the timeout value) and $\text{Unif}_{(0,1)}$ is the PDF of a random variable uniformly distributed between 0 and 1. For simplicity we do not consider the additional round trip delay due to the content request message since it can be correlated with the lookup delay. The CDF of the overall lookup delay, $F_{\text{overall}}(d)$, can be found by integrating Eq. (11.4). Figure 11.4 shows the CDFs of the overall lookup latency for different values of α and β. The input CDF of the round trip delay, $F_{\text{RTT}}(d)$, and the value of $\overline{\rho}_h$ have been obtained by measurements as we will explain in Sect. 11.3. As already observed, by increasing the design parameters α and β, the overall lookup latency increases. The fact that the lookup process and the content search are decoupled results in an overall delay that is strongly dependent on the value of the timeout t.

11.3 Evaluation

In this section we measure the performance of the content management in KAD. We first evaluate the external factors that we cannot influence: p_{stale}, the empirical CDF

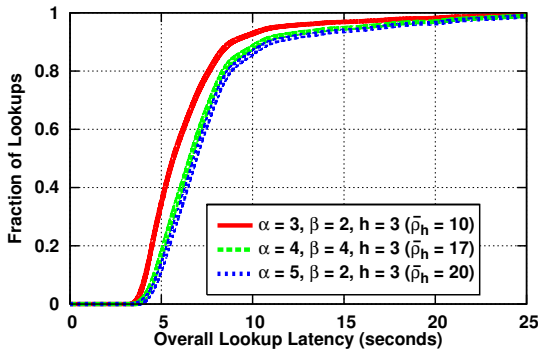


Figure 11.4: Overall Lookup Latency $F_{\text{overall}}(d)$: qualitative analysis ($t = 3$).

of the round trip delay $F_{\text{RTT}}(d)$ and the empirical CDF of the number of hops h. Then we study the current implementation and the impact of α, β, and t.

11.3.1 Measurement Tool and Methodology

For our measurements we have instrumented an aMule client [1] to log all the messages related to content management: route requests and route response, as well as content search and content response. Given a keyword, the client determines the target KAD ID and starts the Procedure `Lookup` and the Procedure `Content Search`. For each message, we register the timestamp, and for lookup responses we register the contacts returned, so that we can evaluate the evolution of the candidate list.

We have extracted 1251 keywords from movie titles found on IMDB [128] and we use them as input for the instrumented client. The hashes of these keywords are uniformly distributed over the hash space. The metrics that can be derived from the collected data are:

p_{stale}: the probability of stale contacts, found as ratio between the number of requests sent and responses received;

CDF of d: empirical cumulative distribution function of the round trip delay for a single message;

CDF of h: empirical cumulative distribution function of the number of hops necessary to reach the target (the first peer that replied with the content);

CDF of the Overall Lookup Latency: empirical cumulative distribution function of the delay between the first *route* request sent and the first *content* response received;

Overhead: The number of route request messages sent during a lookup process.

The initial number of route requests launched is set to $\alpha = 3$; the number of contacts contained in the route response is set to $\beta = 2$. The timeout is equal to $t = 3$ seconds. These are the default values in aMule [1]; we perform a set of experiments by chang-

ing these values and we evaluate the impact of them on the overall lookup latency and on the overhead.

11.3.2 Basic Characteristics

Staleness (p_{stale}).

The first parameter we analyze is p_{stale}, the probability that a contact is stale. We perform the same set of experiments with two different access networks and we have found a value of p_{stale} approximately equal to 0.32. This value has a strong impact on the performance: one third of the contacts are stale, so a lookup process with low α may get stuck with high probability. With the default value $\alpha = 3$ this happens with probability $p_{\text{stale}}^{\alpha} = 0.03$. We will see that this value is partly responsible for the tail of the empirical CDF of the overall lookup latency (see Fig. 11.9).

Round Trip Latency (d).

The other interesting metric that is independent from the client settings is the round trip delay of messages. Figure 11.5 shows the results of our measurement obtained from two different access networks.

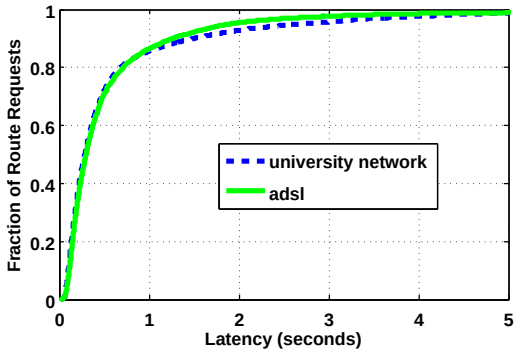


Figure 11.5: Empirical CDF of the round trip delay for route requests.

Almost 80% of the responses are received within 700 msecs after the request was sent. Nevertheless the distribution has a long tail, and this also will have an impact on the overall lookup latency.

Number of Hops (h).

Figure 11.6 shows the empirical CDF of the number of iterations necessary to reach the target. It is interesting to note that at maximum 5 hops are necessary, and in more than 90% of the cases 3 hops are sufficient. This means that, since the KAD network has more than one million concurrent users (Section 9.1.1), the routing tables are very

detailed (about 1,000 contacts). Table 11.1 shows the number of bits gained towards the destination per hop. Initially in average a peer has already nearly one bit in common with the target.

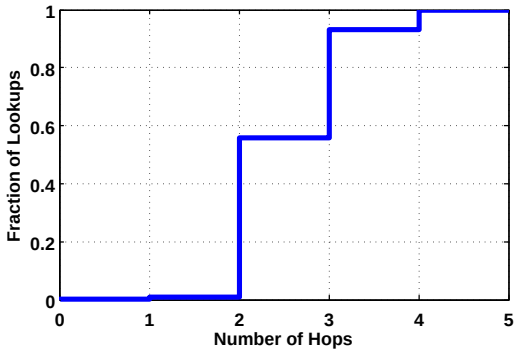


Figure 11.6: Empirical CDF of the number of hops.

hop	bits gained
init	0.98
1	6.13
2	6.02
3	5.24
4	2.30
5	0.00

Table 11.1: The average number of bits gained toward the destination per hop.

Once the content is found, we can evaluate the number of bits in common between the KAD ID of the keyword we searched for and the KAD ID of the contact that replied with the content. This helps in understanding how much the content is spread around the target. Figure 11.7 shows the empirical CDF of the number of bits in common between the replying peer and the content hash. The wide support of the empirical CDF indicates that many keywords can be far from the corresponding target. In Sect. 11.3.4 we will use this observation in order to study the impact of the timeout.

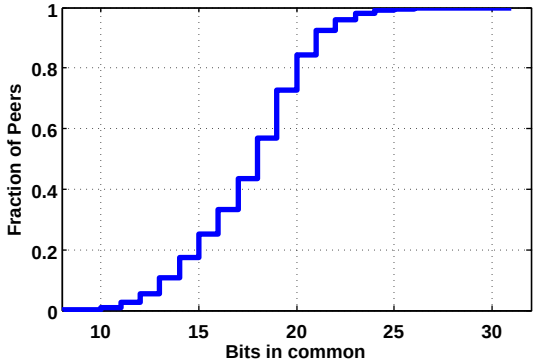


Figure 11.7: Empirical CDF of the bits in common between the peers replying to the search requests with the desired content and the hash of the content.

Actually this is not true for all the keywords. It depends on the popularity of a keyword how widely it is spread in the hash space (Figure 11.8). These results are obtained with an exhaustive search we implemented: first the entire zone around the searched keyword in the hash space is crawled to learn about all peers, second all these peers are queried for the desired content. Rare keywords (as "dreirad" or "fahrrad") have more bits in common with the peers they are stored on, compared to popular keywords (as "the", "french", or "german").

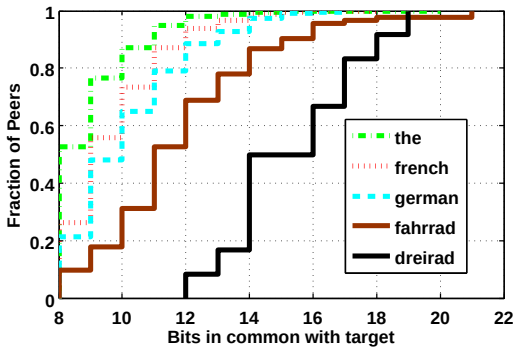


Figure 11.8: The CDF of the bits in common between the peers hosting a content and the hash of the content.

11.3.3 Impact of Different Degrees of Parallelism α

In Figure 11.9 we show the empirical CDF of the overall lookup latency when the parameter α varies from 1 to 7 (its default value is 3). We note a significant difference between the case $\alpha = 1$ and the cases $\alpha \geq 2$, which is due to the high value of p_{stale}. In case of $\alpha = 1$, at each hop only one message is sent; if the contact is stale and the message is lost, the process has to wait for the timeout to expire. This has a strong impact on the overall lookup latency.

With $\alpha = 2$, the probability that the top 2 contacts are all stale decreases significantly. For instance, with $p_{\text{stale}} = 0.32$, the probability that at the first hop both contacts are stale is $p^2_{\text{stale}} = 0.1$. Therefore, the impact of the timeout due to stale contacts on the overall lookup latency reduces, and becomes negligible for $\alpha \geq 3$.

With $\alpha \geq 3$, the different empirical CDFs seems to overlap. If we look in detail at the median (Table 11.2, with $\beta = 2$ and $t = 3$), we see that, as α increases, the median of the overall lookup latency increases. This result was predicted by our qualitative analysis in Sect. 11.2.1 (c.f. Fig. 11.4). The higher α, the more messages the source peer sends ($\overline{\rho}_h$), the longer it takes for the candidate list to stabilize, since it is influenced by the delay of the last received response.

As also shown in the qualitative analysis in Sect. 11.2.1, the support of the empirical CDF starts at $d = 4$ seconds. In the best case, in fact, the candidate list stabilizes after approximately 100 milliseconds (each hop takes at least 40 msec, and the mean number of hops is 2.5). Once the list is stable, the source peer has to wait for the timeout ($t = 3$ seconds), and for the periodic execution of the Procedure `Content`

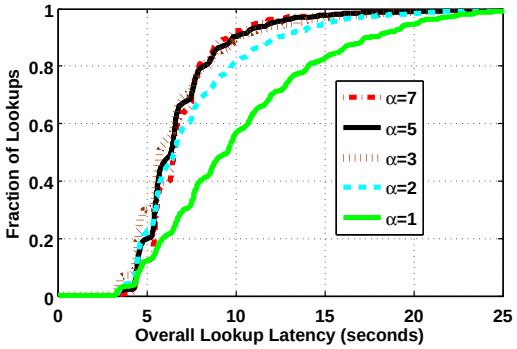


Figure 11.9: Empirical CDF of the overall lookup latencies as a function of the degree of parallelism α ($\beta = 2; t = 3$).

`Search` (in average, 500 msec). If we consider also the application level processing delay, we obtain almost 4 seconds.

As far as the overhead is concerned, Table 11.2 shows the average number of messages sent for different values of α (left hand side of the table, with $\beta = 2$ and $t = 3$). The number of messages sent increases linearly with increasing α.

α	$\beta = 2; t = 3$		$\beta = 2; t = 0.5$	
	average # of messages $\overline{\rho}_h$	median lookup latency	average # of messages $\overline{\rho}_h$	median lookup latency
1	8.5	9.5	10.4	5.6
2	11.5	6.6	12.8	2.4
3	**13.7**	**5.8**	**15.2**	**2.3**
4	16.9	6.1	18.0	2.3
5	20.0	6.4	20.6	2.2
6	22.9	6.5	24.0	2.3
7	26.5	6.5	27.7	1.8
8	30.0	6.6	30.4	1.6
9	32.9	6.6	34.0	1.5
10	36.7	6.6	36.8	2.2

Table 11.2: The overall lookup latency and the number of messages sent per lookup depending on α for different configurations.

11.3.4 Impact of the Timeout t

The default timeout t in aMule is set to three seconds. This implies that the candidate list must be stable for three seconds before the content can be requested. As we showed in Sect. 11.3.2 (Fig. 11.7), the contacts that hold the content may be spread around the target. This means that we could start asking for the the content as soon as the lookup finds a candidate in the tolerance zone, without waiting for the candidate list stabilization.

One possible way to obtain the above result is to decrease the time the Procedure `Content Search` has to wait before starting iterating through the candidate list, i.e., we can decrease the timeout t.

As for α, also t can be changed locally at our instrumented client, without need to update all participants in the network. In Figure 11.10 we show the empirical CDFs of the overall lookup latency for different timeouts for the route request messages.

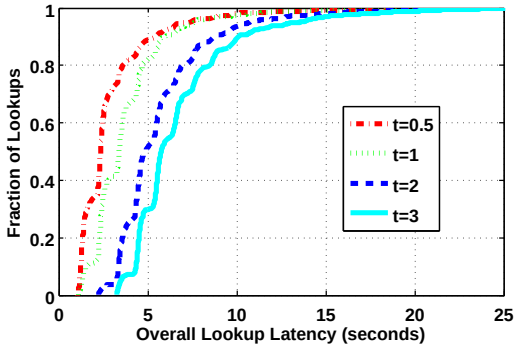


Figure 11.10: Empirical CDF of the overall lookup latencies as a function of the route request timeout t ($\alpha = 3; \beta = 2$).

As the timeout decreases, its influence on the overall latency becomes less significant: reducing the timeout from the default value of 3 seconds to 0.5 seconds decreases the median lookup latency by 60%, from 5.8 to 2.3 seconds. Note that further reducing the timeout would have no effect, since the periodic execution of the Procedure `Content Search` is set to 1 second. Similar results are obtained using a different access network, a common ADSL line (see Fig. 11.11).

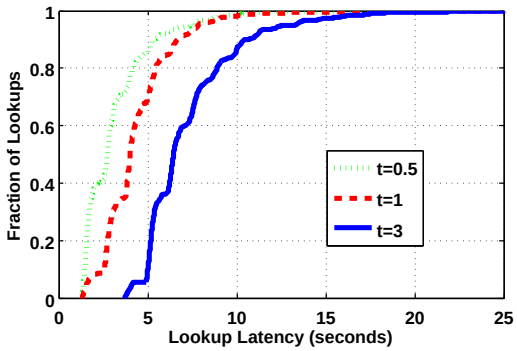


Figure 11.11: Empirical CDF of the overall lookup latencies depending on the route request timeouts for an ADSL client. ($\alpha = 3; \beta = 2; t = *$)

In Table 11.2 we show the overhead for a timeout t set to 0.5 seconds (right hand side of the table). If we compare the default case $\alpha = 3$, $t = 3$ with the case $\alpha = 3$, $t = 0.5$, we see that the overhead is slightly increased: this is mainly due to the the fact that the timeout is also used to trigger new route requests, and, if the responses to the initial α requests arrive later than $t = 0.5$ seconds, new requests are sent out.

Table 11.3: The overall lookup latencies and the number of messages sent per lookup depending on t for different configurations.

t	$\alpha = 3; \beta = 2$		$\alpha = 1; \beta = 2$	
	average # of messages	median lookup latency	average # of messages	median lookup latency
0.5	16.2	2.3	11.4	5.5
1	15.6	3.4	10.9	5.6
2	15.2	4.9	10.2	7.8
3	14.7	5.8	9.5	9.5
4	14.7	7.4	9.1	10.9
5	14.4	8.2	8.7	12.7

In Table 11.3 we show the overhead and the lookup latency for different values of the timeout t. While the median lookup latency decreases significantly for decreasing t the average number of messages sent increases only slightly.

For the peer running at Eurécom a route request timeout set to 0.5 seconds would be optimal, since there is the sweet spot in the distribution of the route request/response RTTs (Figure 11.10). Measurements confirmed that peers being connected to the Internet via a standard ADSL connection do have the same optimum value for the timeout (Figure 11.5). For peers having a poor connection to the Internet this value needs to be adapted. We suggest to keep in memory a running average of the RTT as done in TCP and adjust the timeout accordingly. In other applications the timeout is usually set to two or three times the RTT. Since this application does not depend on a specified message, but the goal is to decrease the lookup time, it makes sense to set the timeout tighter. Some messages may come in late, but the approach toward the targets goes on.

11.3.5 Impact of the Number of Contact Asked For

Once observed the gain that can be obtained by eliminating the effect of the timeout, we want to evaluate the impact of the parameters α and β on the overall lookup delay. Recall that β is the number of closer contacts that are asked for by a route request message. Unfortunately, this parameter cannot be chosen freely in the source code, but can be only set to 2, 4, or 11. We performed measurements for $\beta = 4$ and varying α.

Figure 11.12 shows the results we obtained with different settings. The more messages we send, the more the overall lookup latency is reduced. This comes at a cost of increased overhead. For instance, for $\alpha = 5$ and $\beta = 4$ the mean number of messages is equal to 29. By increasing further the values of the parameters, we are not able to notice a significant improvement in the empirical CDF, since the periodic execution of the Procedure `Content Search` determines the overall lookup delay.

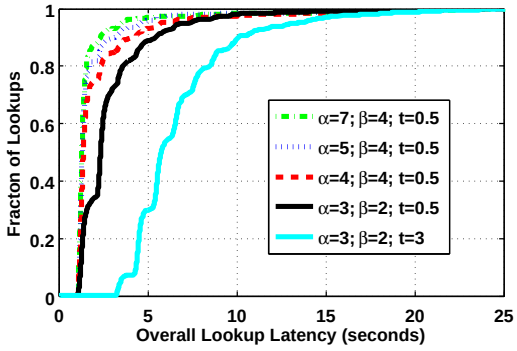


Figure 11.12: Empirical CDF of the overall lookup latencies varying α and β.

11.4 Improving the Content Lookup

The evaluation of the impact of the timeout t on the overall lookup latency suggests that a different approach to the content management process would bring a substantial gain.

The idea is to differentiate the software architecture according to the two different aims – publishing or retrieval. For content publishing, the main issue is the role of the timeout in the stabilization of the candidate list: a timeout occurs when candidates do not reply or when candidates reply with contacts that are not closer than the other candidates. The former increases the total delay and should be considered separately. The solution is then to decouple the timeouts for the two different situations. At this point, we believe that the approach to wait until the list is stable before publishing (because it contains the best candidates possible) is the best one.

Instead, for the content retrieval, this process should be strictly coupled with the lookup process: as soon as the lookup finds a candidate in the tolerance zone, a content request should be sent. We call this approach *Integrated Content Lookup (ICL)*. In Sec. 11.3.4 we have shown how to obtain a similar objective with a simple *hack* of the code: by decreasing the timeout t we let the content search process iterate though the candidate list more frequently. The results we have obtained are pessimistic, since they include the delay due to the periodic execution of the Procedure `Content Search`. In this section we propose a model that shows the qualitative performance in terms of the overall lookup latency of ICL scheme.

We assume that the probability that all the initial α contacts are stale, $p^{\alpha}_{\text{stale}}$, is negligible. Among the initial α messages, only $\alpha(1-p_{\text{stale}})$ replies are received. The process continues to the next hop using the contacts contained in the first reply. Thus, the delay of the first hop is the *minimum* delay among the replies. It is simple to show that the corresponding CDF for the first hop is equal to (see [19], Eq. 2.8)

$$F_{1,\,\text{ICL}}(d) = 1 - [1 - F_{\text{RTT}}(d)]^{\alpha(1-p_{\text{stale}})} \,. \tag{11.5}$$

At this point we neglect the contacts contained in the route responses that come after the first, and concentrate only on the β contacts we received. This simplification ignores possible better contacts contained in the responses of the first hop that are

received later: in this sense the analysis is conservative. We assume that the contacts contained in the first response are placed in the top of the candidate list (they are closer to the target than the candidates already present). In the second hop the process sends $\gamma = \min(\alpha, \beta)$ new route requests. Among them, only $\gamma(1 - p_{\text{stale}})$ replies are received. The CDF of the delay for the second hop is

$$F_{2,\,\text{ICL}}(d) = 1 - [1 - F_{\text{RTT}}(d)]^{\gamma(1-p_{\text{stale}})} \,. \tag{11.6}$$

For the following hops, we have the same behavior as for the second one. When a contact replies, the Integrated Content Lookup process checks if it falls in the tolerance zone and immediately send a route request. Thus, the overall lookup latency is given by the sum of the delay of the single hops. Let $f_{\text{i},\,\text{ICL}}(d)$ be the PDF of the delay for a single hop i, i.e., the derivative of $F_{\text{i},\,\text{ICL}}(d)$ of Eqs.(11.5) and (11.6). The PDF of the overall lookup latency, $f_{\text{ICL}}(d)$, is then

$$f_{\text{ICL}}(d) = f_{1,\,\text{ICL}} * f_{2,\,\text{ICL}} * \ldots * f_{\text{h},\,\text{ICL}}(d) \tag{11.7}$$

where the convolution is done for all the h hops. The CDF of the overall lookup delay can be found by integrating Eq. (11.7). Figure 11.13 shows the CDFs of the overall lookup latency for different values of the parameters α and β, with a number of hops $h = 3$. We consider the input CDF of the round trip delay, $F_{\text{RTT}}(d)$, shown in Sect. 11.3. The design parameters α and β now have a significant impact on the overall lookup latency, at a cost of increased overhead. This qualitative analysis yields the same results as shown in the experimental evaluation, where we studied different settings for the parameters α, β and t (Fig. 11.12). It is interesting to note that the CDF has a similar tail as we found with measurements: this means that the tail of the input CDF $F_{\text{RTT}}(d)$ has a strong impact, even for large α and β.

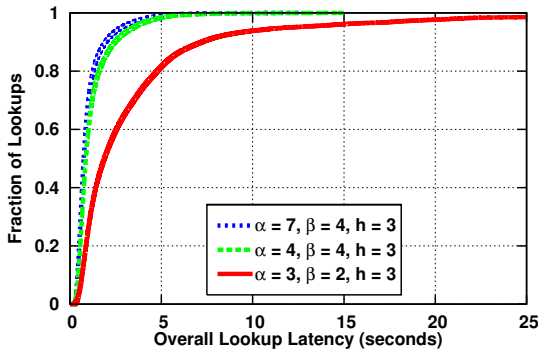


Figure 11.13: CDF of the Overall Lookup Latency, $F_{\text{ICL}}(d)$: qualitative analysis.

11.5 Related Work

Stutzbach and Rejaie [122] did a detailed analysis of the routing tables and the lookup process in KAD and provide latency measurements for varying α. However the two remaining parameter β and t are not mentioned.

Falkner at al. [40] analyzed the implementation of KAD in Azureus. They measured a value of 127 seconds (more than 2 minutes!) for the median of the overall lookup latency. The measurement we did on KAD using the default configuration showed a median overall lookup time of 5.8 seconds.

Dabek at al. [27] briefly describe the iterative lookup used by Chord and Kademlia, and perform a simulation using latency data obtained by the King method [48] (where the average one hop latency is of 154 milliseconds), but only for Chord.

Li at al. [70] describe the lookup process in KAD identifying the parameters α and β. However they state that the lookup process tries to keep always α requests open. In contrast, we have found that the number of request can vary between 0 and $\alpha\gamma$. Moreover, they do not analyze the impact of the timeout t. Finally, they performed a simulation with 1,024 nodes of the overall lookup latency using p2psim and one hop latency data obtained of with the King method [48], finding an average overall lookup latency of 250 milliseconds.

11.6 Discussion and Conclusions

We study the content management process implemented in KAD and we show that the design approach has a strong impact on the overall lookup latency in case of content retrieval. We perform measurements with two different objectives: (i) we characterize the external factors that influences the performances – such as the probability that entries in the routing tables are stale, or the round trip delay of messages; (ii) we evaluate the influence of the design parameters – such as the number of requests sent initially or the timeout – on the performance.

We show that, by coupling the lookup procedure and the the content retrieval process, it is possible to decrease the overall latency while keeping the same overhead. The results we obtain suggest that the scheme can be further improved if we let the design parameters to change, i.e., if we make them adaptive. For instance, in contexts with a low churn, the probability that entries are stale (p_{stale}) reduces, and thus it is not necessary to have a large degree of parallelism in the sent requests. In this case we may choose α and β as functions of p_{stale}, rather than simply taking fixed values. The same applies for timeout t, i.e., we may choose t as a function of d, i.e., the (estimated) round trip delay.

CHAPTER 12

Applying the Developed Measurement Techniques to Azureus

In this Chapter, we present the results we obtained by crawling the DHT of the BitTorrent client AZUREUS for several months. We use the crawling technique presented in Section 8.3.

To crawl the DHT implemented by a BitTorrent client is a new approach to learn about the behavior of BitTorrent peers, since to this day the classical approach was to learn about torrents on portal web sites and to contact the corresponding trackers then. Since in BitTorrent every torrent forms a swarm of peers, all existing torrents need to be tracked in order to get a full view on all clients participating. Our approach is not based on torrents, but we make use of the DHT in which all peers participate and in which information about all torrents is published. For the moment, this DHT is used as a fall back mechanism in case the tracker goes down. Not all DHTs implemented by the different BitTorrent clients are compatible to each other. Therefore, with our method of crawling a DHT, we do not learn about all peers using BitTorrent, but only about those using a DHT compatible to the DHT of AZUREUS. There are several incompatible implementations of DHT by different BitTorrent clients. However, keeping track of a handful DHTs is more feasible than keeping track of hundred of thousands of torrents, knowing that only those published on websites can be tracked.

As in KAD, each AZUREUS node has a global identifier in the DHT. However, the ID is not chosen randomly. First, a string is built up out of the IP address and the port number mod 1999, which is a prime, separated by ":". The SHA1 hash of this string is the DHT ID of the node. Thus the number of AZUREUS nodes per IP address is limited to 2000, since a client running on port 1999 has the same ID as a client running on port 3998 on the same IP address. Moreover, since most peers do have assigned a new IP address by their provider, every time they reconnect to AZUREUS they will change their identifier in AZUREUS. The bucket size in AZUREUS is with 20 peers twice as large as in KAD. Another major difference to KAD is that no search functionality is available to the end user. This is mainly due to the fact that AZUREUS

does not have a two level publishing system like KAD (cf. Section 8.2). This makes it impossible to search for a keyword. The search function does only allow to search for a torrent once it is downloaded from a web site, and the tracker for this torrent is not available anymore.

The AZUREUS peers use the network coordinate system Vivaldi [26] (cf. Section 8.6) to compute their network coordinates. With the help of these coordinates, they try to find other peers physically close to them in order to reduce download times and to keep the traffic local, e.g. inside an ISP or a country.

By crawling AZUREUS, we collect round trip times at the application layer and at the network layer of several hundred thousand of clients around the world. We also collect their Vivaldi network coordinates and analyze their accuracy.

12.1 Measurement Methodology

The first step in order to do any latency measurements is to learn about the hosts you want to measure up to. Therefore, we used our crawler *Blizzard* for KAD (Section 8.3) and adapted it to work for AZUREUS as well. Our crawler logs for each peer P

- the time of the crawl
- the IP address of P
- the port number used for the control traffic
- the Vivaldi network coordinates (version 1 with 2 dimensions, the height vector, and the estimated error, and version 2 with 4 dimensions, the height vector, the estimated error, and the age of the coordinate) [26, 65]
- the estimated number of peers in the network. The number of participants is estimated by computing the peer density in a small fraction of the DHT and interpolating this number.

12.1.1 Application Layer Round Trip Time

The crawler exploits the control messages of the DHT, it uses the messages intended for routing, to learn about other peers. The *application layer round trip time* (ARTT) of those messages is composed by two parts: the network layer round trip time (NRTT) and the additional delay induced by the computation of the application that depends on the load of the end-system and of the access link. Therefore, we always expect the application layer round trip time to be larger than the network layer round trip time. In the remaining of this chapter, the round trip time is always expressed in milliseconds.

12.1.2 Network Layer Round Trip Time

The REPLY_FIND_NODE message contains the IP address and the port of a peer. After the discovery of a peer, a TCP packet with the ACK flag set is sent on this very port. The peer is expected to reply with another TCP packet having the RST flag set [93]. We call the delay between the emission of the TCP ACK and the reception of the TCP RST the *network layer round trip time* (NRTT). The TCP ACK packet we send is directly processed by the kernel of the operating system of the queried peer. Therefore, we expect the network layer round trip time to be equal to the round trip time of an ICMP ping. We send a TCP ACK instead of an ICMP ping since most (Wlan-) routers and personal firewalls do not reply to ICMP pings. Moreover, by default all ports a closed, which is why it is so important to learn about an open port by first crawling the peer-to-peer network.

12.1.3 Network Coordinates

The reply messages received during a crawl also contain the Vivaldi [26] network coordinates of the queried peer. Depending on the version of the AZUREUS client, different versions of the Vivaldi network coordinates are communicated: none, version 1 (made of 2 dimensions plus the height), or version 1 and version 2.4 (made of 4 dimensions plus the height). The difference between the two implementations of the Vivaldi network coordinate system is not limited to the number of dimensions, also the age of the coordinate is transmitted. Moreover, the ways new measurements are used to update the coordinates are much more sophisticated. In total, 16 additional bytes are transmitted for version 2.4. See [65] for details about the different versions of the network coordinates. 97.2% of the AZUREUS clients use the latest version of the DHT protocol that transmits both, version 1 and version 2.4 of the network coordinates. Therefore, we considered only these peers.

We run an AZUREUS client on the machine performing the crawl to learn about the network coordinates of the crawler itself. Using those coordinates, we are able to compute the Euclidean distance between the crawler and the queried peers in the network coordinate system. It is expected to approximate the round trip time of the packets sent to this peer. Since the AZUREUS application is not aware of the NRTT, but only of the ARTT, we expect the Vivaldi distance to be more tightly correlated with the ARTT than with the NRTT.

12.2 Dataset

For the analysis presented in this chapter, we collected the following datasets.

1. **Mannheim:** From the University of Mannheim, attached to the German research network. One single crawl performed on March 31, 2008, at 08:00

CET. 1,044,155 peers have been discovered, 291,850 responded to the crawler (AZUREUS ARTT and network coordinates are available). 157,205 peers also replied to the TCP AKC. For those peers, the full data is available. The crawl duration was 12 minutes.

2. **Eurécom:** From Institut Eurécom, attached to the French research network. Starting on 15th of February, 2008, we performed 3 full crawls of the AZUREUS network a day (05:00, 13:00, and 21:00 CET). On each crawl, 1 – 1.4 million were discovered. About 300,000 – 400,000 of them responded to the crawler, thus, of those peers, AZUREUS ARTT and network coordinates are available. For about 50% of the responding peers, the NRTT is also available, the other peers did not respond to the TCP ACK packet sent. Each crawl has a duration of about 20 minutes.

3. **ADSL:** From a France Télécom ADSL line. One single crawl was performed on March 26, 2008. This crawl took 12 hours and, out of 1,267,822 discovered peers, 118,548 peers responded. Whereas the NRTTs are only available for 37,346 of those peers.

12.3 Crawl Results

In this Section, we present the remaining data collected while crawling the peers of AZUREUS, such as their uptime, their version, their geographical origin, their service providers, the ports used, the overhead due to the DHT maintenance, the estimated number of peers in the network, and, finally, their number of contacts.

12.3.1 Uptimes

Every peer counts the time elapsed since it has been launched. The median uptime amounts to 8 hours, 5% of the peers have been up longer than 6 days and one 1% longer than 26 days (Figure 12.1). This is very long compared to a median of 2 1/2 hours for the session times in eMule and aMule (Section 9.2).

The distribution of the uptimes is heavy tailed since it decays slower than exponentially. We refer to a distribution as heavy-tailed if its coefficient of variation is larger than 1, the one of the exponential distribution. The coefficient of variation ($\frac{\sigma}{\mu}$) of the uptimes is 14.5.

12.3.2 Versions

In total, we saw 191 different versions of clients during our crawls. In Figure 12.2, the 10 most frequently used version on April 25, 2008, 13:00 CET are plotted. Note that version 3.0.5.2 is the most up to date version. In fact, the latest version of the

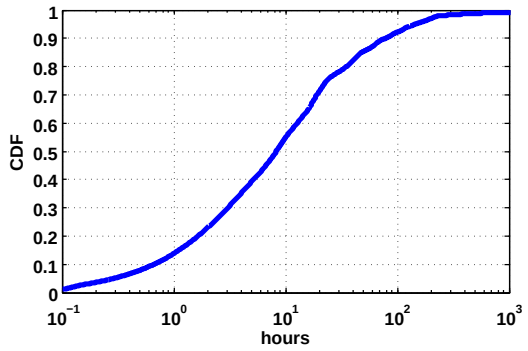


Figure 12.1: CDF of the uptimes of the AZUREUS clients.

AZUREUS client automatically updates itself, which is why more than 75% of the peers use this version. The automatic update is able to reach most of the users running a version including the automatic update within only a few days. In Figure 12.3, one example of a version change is plotted. On March 5, 2008, the new version 3.0.5.0 was introduced and replaced the version 3.0.4.2 within a few days. You may notice that until April 25th another version, 3.0.5.2, was introduced, which almost completely replaced its predecessor.

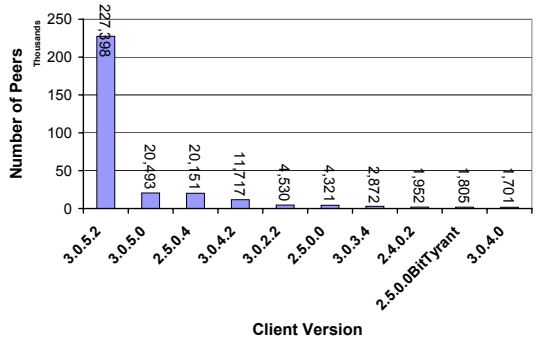


Figure 12.2: Most used versions on April 25, 2008.

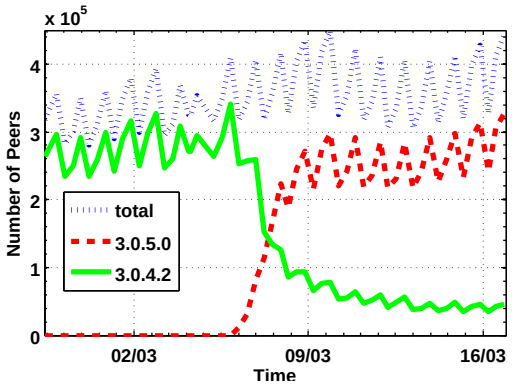


Figure 12.3: Quick Rollout of a new version.

The DHT protocol itself has different versions, too. 97.2% of the clients use the latest version 16 (VIVALDI_FINDVALUE), 2.2% use the version 15 (GENERIC_NETPOS),

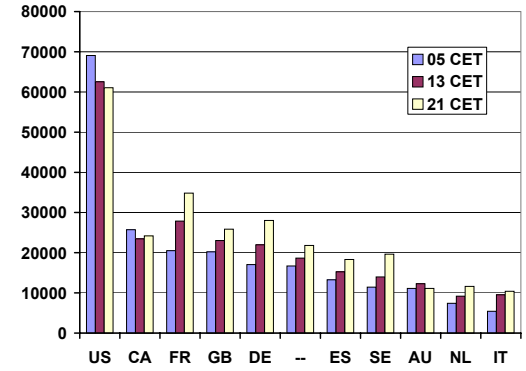
and 0.6% use the version 14 (VENDOR_ID and BLOCK_KEYS). Version 14 does not transmit any Vivaldi coordinates, version 15 transmits the old version V2.2 of the Vivaldi 2 coordinates (the most recent version is V2.4), this version uses 5 dimensions and not 4 dimensions plus a height vector.

12.3.3 Geographical Origin

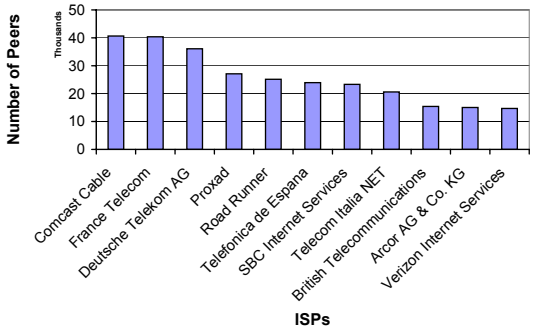
Peers from the US dominate the AZUREUS network. They are followed by peers from Canada and Western Europe. In total, peers from over 190 different countries have been discovered. Figure 12.4(a) shows the distribution of the most important countries on April 25, 2008, at different times of the day. Note that most users use AZUREUS in the evening hours.

A histogram of the most used Internet providers can be found in figure 12.4(b). According to their countries of origin, the biggest providers are found in the US, France, Germany, Spain, and Italy.

The user's behavior seems to be similar to the one observed in KAD. However, in KAD, Chinese and European peers dominate, peers from the US don only play a negligible role (Section 9.1.1).



(a) Geographical origin of the AZUREUS peers.



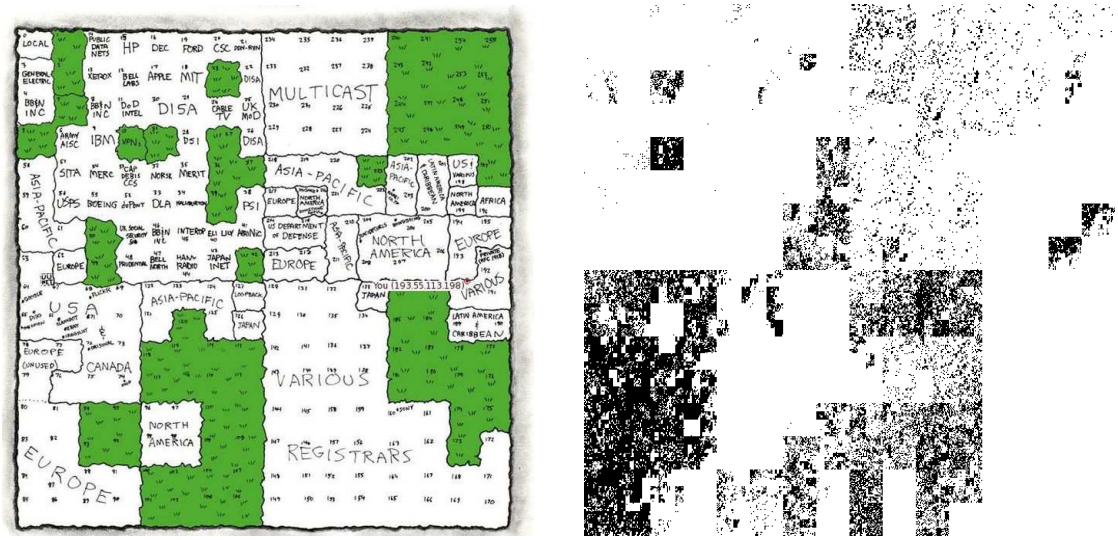
(b) Internet provider of the peers.

Figure 12.4: Country and ISP of origin.

12.3.4 IP addresses and ports

To get an idea where the AZUREUS users come from in the Internet, we plot the IP addresses on the *Map of the Internet* from the xkcd comic (Figure 12.5). The comic orders IP addresses on a Hilbert curve and marks the /8 blocks of the IP addresses by their original allocation. Any consecutive string of IP addresses will translate to a single compact contiguous region on the map. Each of the 256 numbered blocks represents one /8 subnet.

Most of the users come from an IP space allocated to the US, Canada, Europe, and various registrars. However, there are some users from class A networks belonging to BBN, MERC, and others.



(a) Address space (b) IP origin of AZUREUS peers

Figure 12.5: Map of the Internet

Only about 5% of the peers make use of the default BitTorrent port 6881. The other peers use randomly assigned port numbers above 50,000 (Figure 12.6).

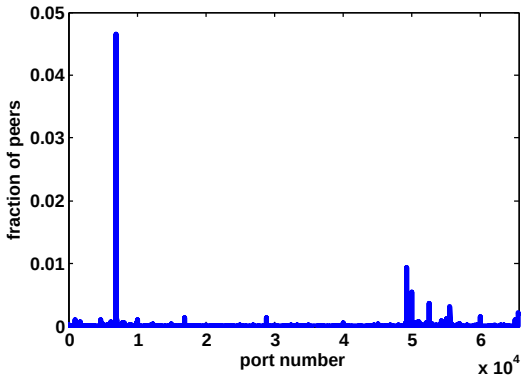


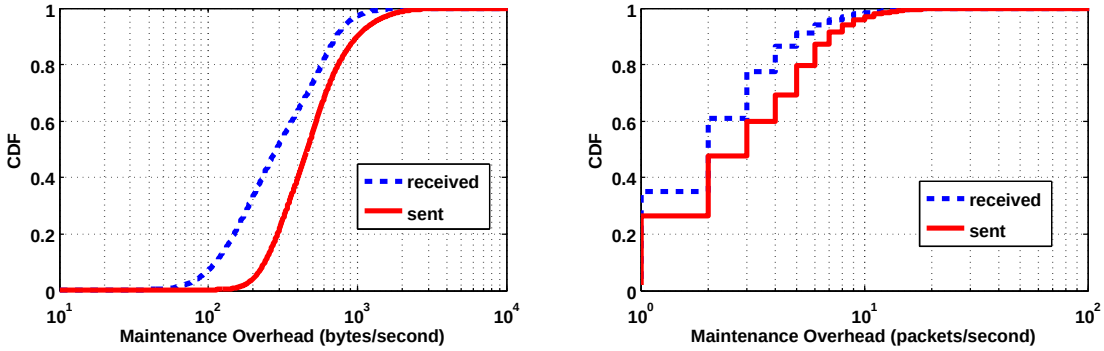
Figure 12.6: Histogram of the ports used by the AZUREUS peers.

12.3.5 Overhead

The statistics also include information about the overhead needed to maintain the DHT. The median amount of data a peer receives is 286 bytes (2 packets) per second and the median amount of data sends is 462 bytes (3 packets) per second (Figure 12.7). For comparison, the data of a KAD client that we obtained by dumping all the communication needed for the maintenance of the DHT over 24 hours: It summed up to 4 packets per second or 422 bytes per second, which is less compared

to AZUREUS. One reason for this additional overhead may be the double bucket size since all these peers need to be regularly checked for availability.

The difference between the amount of packet overhead sent and received is due to the fact that some messages are sent to stale peers. These messages are accounted on the sending side, but they are never received by anyone. From the collected data, we can deduce that about one third of the contacted peers are stale. This is about the same number we obtained for KAD (cf. Section 11).



(a) Bytes per second (b) Packets per second

Figure 12.7: Maintenance overhead

Having a look at the different message types, one can notice a huge difference to the values we found in KAD (Section 8.4). In AZUREUS, the median number of store requests per second is 0.07, the median number of find value request is 0.31 (Figure 12.8). There are 4 times more find value requests than store requests. In KAD, this is the opposite. There are 10 times more publish requests than search requests. This is due to two main facts: First, in KAD, keyword and sources are published, whereas AZUREUS has only one layer of metadata. Given that per source there are 5 keywords on average, the amount of messages needed for the publication of a source is multiplied by 6. Second, the republishing interval of a content in AZUREUS is of 8 hours, whereas sources in KAD are republished every 5 hours.

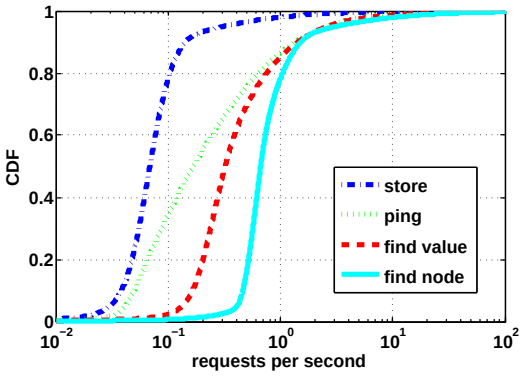


Figure 12.8: CDF of the different requests received per second per node.

The median number of entries of the DHT a node stores in its local hash table is 402(Figure 12.9). Long living nodes store up to ten thousand values, the maximum we found was 34,807.

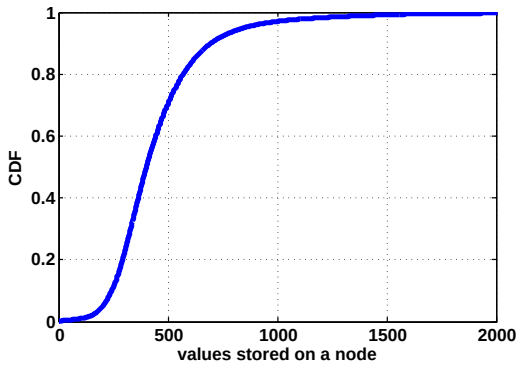


Figure 12.9: CDF of the number of entries stored per node.

12.3.6 Estimated number of peers

Every peer participating in the AZUREUS network estimates the number of peers in the network. In this section, we first describe the algorithm for the estimation of the number of peers, second we evaluate its accuracy by comparing the estimates with our measurements.

The estimation of the size of the DHT implemented in AZUREUS is based on a density function. N_0 is the node itself, N_1 the nearest peer, N_2 the 2nd nearest peer ... N_p the p_{th} nearest peer in the routing table of N_0. p is set to 20 since there are at most 20 peers in one bucket of the routing tree. In this case, it is the deepest bucket containing the closest contacts. Let D_i be the XOR distance between N_0 and N_i. The local estimation of the DHT size n is done in the following way:

$$n = \frac{2^{160}}{\frac{\sum_{i=1}^{p} i\, D_i}{\sum_{i=1}^{p} i^2}}$$

The numerator is the maximum possible number of nodes in the DHT, the denominator is the filling level of the hash space. It is equal to 1 if every position is filled, i.e., if every hash value is occupied. The term $i\, D_i$ in the numerator of the denominator is composed of i that stands for the ith closest peer and D_i for its distance. Since the ith closest node needs to have at least a distance of i to N_0, in case every place in the hash space is taken, the maximum value this term can take is i^2, the term we find in the denominator.

Consider the case of $p = 2$, the first node has a distance of 2 to N_0, there is a space for only one other node in between, the second node has a distance of 4, there is only one possible space between the 2nd node and the 1st node. In other words, every second

possible place is taken. In this example, the denominator would be $\frac{1*2+2*4}{1+4} = 2$; the estimation for the number of nodes is going to be 2^{80} since every second place in the hash space is empty.

In a second step, this value is averaged with the estimation learned from the other nodes except for the three smallest and the three biggest values.

For one crawl, we evaluated the accuracy of the estimation of the number of peers. Our crawler found 1,040,943 peers, we queried each peer for its estimate of the network size. The average estimation was 1,101,500, the median 1,097,223. The minimum was 2, which is most probably a peer still in the bootstrap phase, i.e., the routing table is not yet filled, the maximum was 12,461,686. In most cases, however, the estimation is very accurate, the deviation compared to the number of peers we found by crawling is less than 5%.

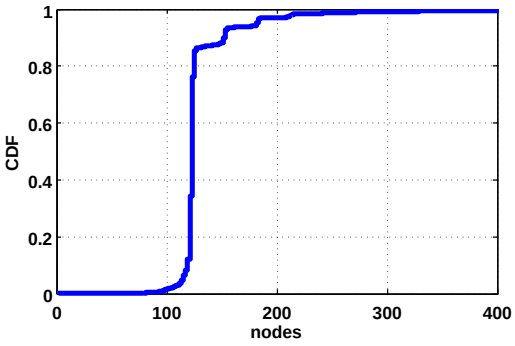
12.3.7 Contacts

The contacts in AZUREUS are organized in a routing tree. The median of the number of nodes in the tree is 123 (75% of the peers have values between 119 and 125), for the number of leaves (buckets) attached to the tree it is 63 (75% of the peers have values between 60 and 63). The median of the number of contacts in the routing tree is 1,107 (80% of the peers have values between 1,050 and 1,155) (Figure 12.10). This is about the square root of the number of participants $O(\sqrt{n})$, much more than the proposed DHT routing table size in literature, e.g. Chord, that grows logarithmically with the number of participants $O(\log(n))$.

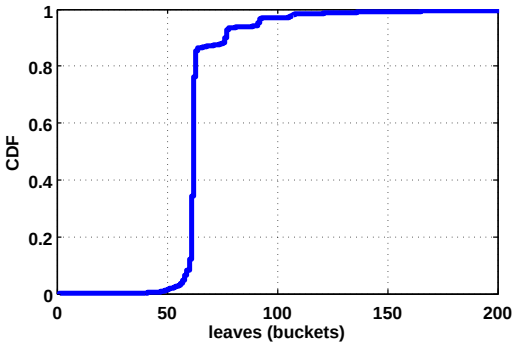
Calot [126] ensures 2 hop routing to any destination with a routing table of $O(\sqrt{n})$ contacts, which is slightly better than the number of hops needed by AZUREUS.

12.4 Evaluation of the Vivaldi Internet Coordinates Used in Azureus

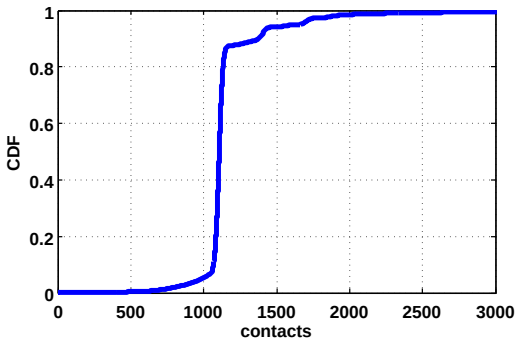
Table 12.1 gives an overview of the results obtained on AZUREUS in the Mannheim dataset. The results of the two other vantage points are omitted for space constraints since they are qualitatively very similar. We mapped the IP addresses of the peers to their countries using Maxmind [76], a database containing geo-location information. In the table, we list several countries from different parts of the world that are representative for their continents: countries close to our crawl site, countries far away, countries with good and with poor Internet connectivity. The second column shows the number of unique AZUREUS clients measured, the third column shows the average ARTT followed by 5 columns for the NRTT. Columns 9 and 10 show the average Euclidean distance from our crawl site to the AZUREUS peers.



(a) CDF of the number of nodes in the routing tree.



(b) CDF of the number of leaves (buckets) in the routing tree.



(c) CDF of the number of contacts in the routing tree.

Figure 12.10: Data about the routing tree.

The mean NRTT value does not allow to distinguish between continents, compare 566 ms for Spain with 319 ms for the US or 377 ms for Korea. In fact, the *mean* NRTT is strongly biased by outliers, as can be seen from the 95th percentile (Column 8 and figure 12.11). A much better indicator for geographic proximity is the 5th percentile: 21 to 57 ms for European countries, 111 to 120 ms for North America, 169 to 233 ms for South America, and 281 to 340 ms for Asia and Australia. Even if it is possible to make a difference between the NRTT distributions for different continents, this does not imply that, from the NRTT to a single peer, one can deduce its continent of origin. We do explain the very low variance for peers in Korea with the widely deployed fiber to the home in these countries. The high variance of the NRTTs, e.g. in Europe, is introduced by the last mile to these users that are often connected with ADSL [32]. The ADSL access links have buffers that can add an additional delay ranging from tens of milliseconds to more than a second.

Compared to the mean NRTTs (Column 4), the mean ARTTs (Column 3) are slightly higher and they show a higher variance. This is the additional delay introduced by the AZUREUS application. However, the overall shape of the corresponding cumulative distribution function per country remains the same. In figure 12.11, the measured CDF of the NRTTs are plotted.

1	2	3	4	5	6	7	8	9	10
Country	# Clients	ARTT	NRTT					Coordinates	
		mean	mean	st. dev.	5th perc.	median	95th perc.	v2	v1
France	13,775	359	306	626	*44*	92	1,235	542	344
Germany	11,439	435	236	598	*21*	64	1,143	736	415
Spain	8,281	641	566	984	*55*	125	2,604	1,043	581
Italy	3,464	389	325	671	*57*	119	1,286	560	368
Canada	12,349	360	298	512	120	169	948	454	259
US	32,528	394	319	543	111	176	1,052	488	269
Venezuela	530	851	765	1,175	169	258	3,299	1,197	657
Brazil	2,364	776	718	1,067	233	312	2,828	1,053	598
China	315	563	513	302	324	413	1,101	656	381
Korea	362	413	377	134	308	346	*563*	372	231
Japan	1,283	443	370	358	281	300	*586*	466	246
Australia	3,733	934	872	1,326	340	392	3,729	1,162	634
All countries	157,205	454	375	739	37	151	1,541	647	376

Table 12.1: Overview of the AZUREUS results: ARTTs, NRTTs, and coordinate distances in milliseconds. Measurement host is located in Mannheim.

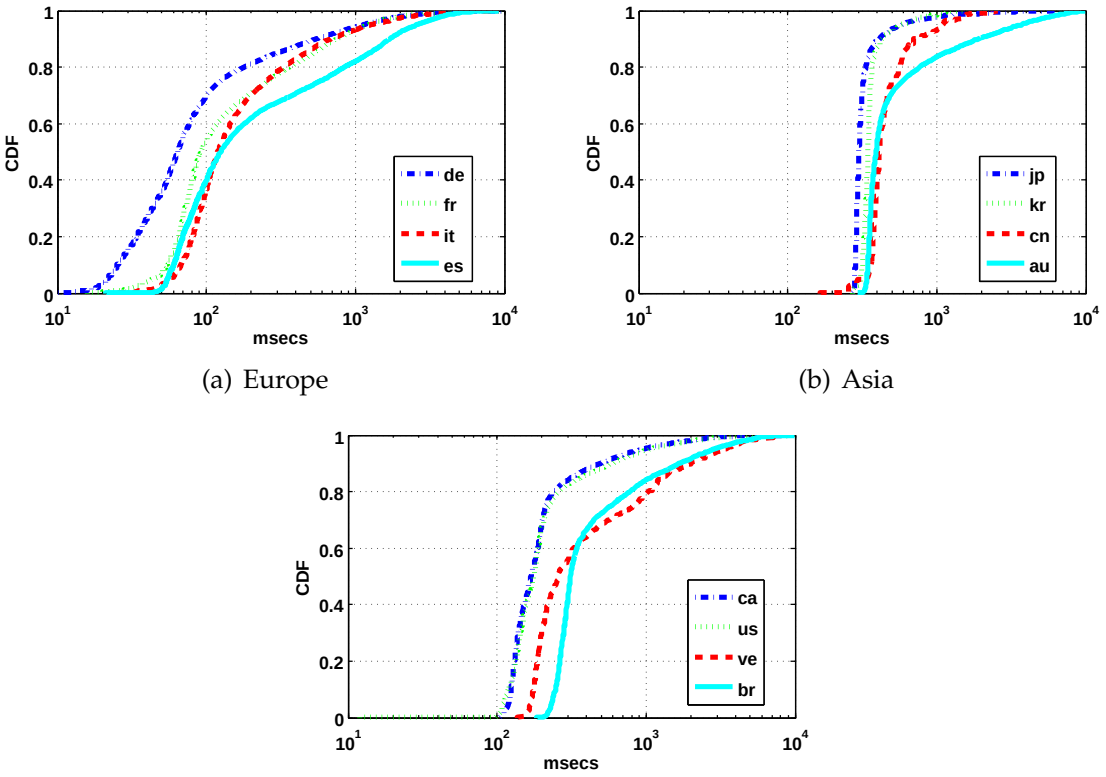
12.4.1 Network Coordinates

We compared the calculated Euclidean distances for both implementations of the coordinate system to the application round trip time measurements and to the network layer round trip time measurements we performed.

1	2	3	4	5	6	7
	ARTT NRTT	v.1 v.2	v.2 ARTT	v.2 NRTT	v.1 ARTT	v.1 NRTT
France	0.94	0.89	0.91	0.89	0.85	0.84
Germany	*0.65*	0.91	0.92	0.63	0.87	0.61
Spain	0.93	0.94	0.94	0.89	0.91	0.87
Italy	0.90	0.92	0.91	0.89	0.85	0.84
Canada	0.85	0.81	0.83	0.81	0.77	0.74
US	0.81	0.86	0.85	0.82	0.79	0.74
Venezuela	0.97	0.84	0.93	0.94	0.79	0.80
Brazil	0.94	0.90	0.92	0.93	0.87	0.88
China	0.75	0.80	0.93	*0.54*	0.77	0.63
Korea	0.94	0.74	0.83	0.80	0.60	0.59
Japan	*0.48*	0.90	0.93	0.47	0.93	0.47
Australia	0.98	0.87	0.91	0.91	0.85	0.84
All countries	0.88	0.89	0.90	0.84	0.84	0.80

Table 12.2: Correlations of the results for RTTs and Euclidean distances shown in table 12.1.

The ARTTs and the NRTTs do have a positive linear correlation of 0.88 (Table 12.2, column 2). For some countries, such as Germany and Japan, the correlation between the ARTTs and the NRTTs is much lower compared to other countries. Therefore, the correlation between the ARTTs and the distances computed with the coordinates are



(a) Europe

(b) Asia

(c) America

Figure 12.11: NRTT for different countries. Origin of the measurements is in Mannheim / Germany.

lower, too. The weak correlation for the German peers is not due to the measurement origin being in Germany, the other two datasets also confirmed these results.

The correlation between the two versions of the network coordinates is of 0.89 (Column 3). The correlation between the coordinates version 1 and the ARTTs is 0.84 (Column 6), for version 2 this increases to 0.90 (Column 4). The correlation between the network coordinates version 1 and the NRTT is 0.80 (Column 7), for version 2 this value increases to 0.84 (Column 5). We can conclude that the two additional dimensions, the introduced age of the coordinates and the resulting additional overhead in version 2 do not result in a significant improvement of the accuracy of the network coordinates.

Our direct measurements of the NRTT are all taken from hosts based in Europe, thus the CDFs for the different countries do all have Europe as a point of origin (Figure 12.11). To get a different point of origin, we need to make use of the *network coordinates*. We chose an AZUREUS client in the US and computed its Euclidean distance to all other peers. In Figure 12.12, the Vivaldi distances of this US peer to peers in Japan, Canada, and Germany are plotted. Surprisingly, Japan seems to be closer to the US than Canada. Thus, based on the distance computed using the network coordinates, a peer located in the US would prefer Japanese over Canadian peers.

The CDFs of the coordinate distances of a peer based in Germany have a shape similar to the CDFs of the NRTTs shown in Figure 12.11.

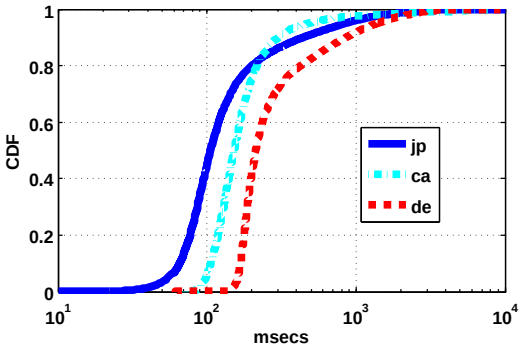


Figure 12.12: Vivaldi distances for different countries. Origin is in the US.

12.4.2 Height

The height value in the network coordinates should reflect the latency introduced by buffers on the last mile toward the end-user. The usage of the height is necessary in order to not distort the coordinate system by large latencies introduced on ADSL lines. We extracted all peers from our dataset that are customers of the provider France Télécom and that are in Nice / France. The propagation delay between any pair of those clients is in the order of a few milliseconds.

Using the Vivaldi coordinates of those peers, they should all be very close to each other if the height value is discarded given the small geographic distances between those hosts. Considering the height, however, they can be far away from one to another. In figure 12.13, the CDF of the pairwise distances is plotted. We see that the latency introduced by the ADSL links is completely reflected in the height value and not in the coordinates. For version 2, the results are even better than for version 1, the distance ignoring the height is only of 38 milliseconds in median.

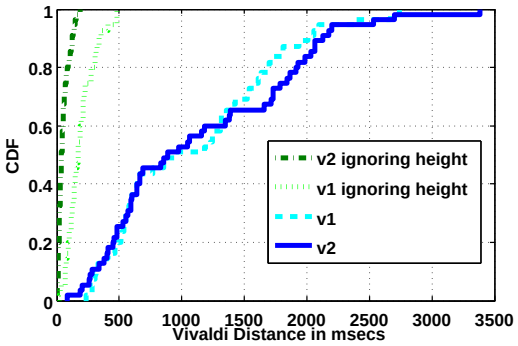


Figure 12.13: CDF of the pair wise distances of the clients of France Télécom in Nice / France, with and without considering the height value of the coordinates.

Figure 12.14 shows the pairwise Vivaldi distances of all peers in France. Again, the distances without the height do reflect the geographic distances, whereas the dis-

tances including the height are of one order of magnitude larger due to the queuing delay on the last mile.

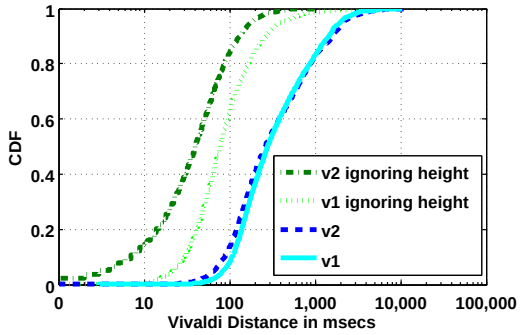


Figure 12.14: CDF of the pair wise distances of the clients in France, with and without considering the height value of the coordinates.

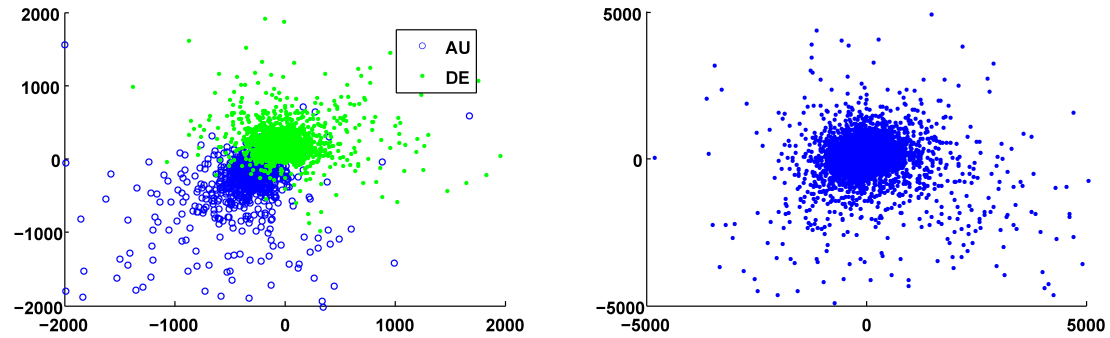
12.4.3 Visual Check of the Network Coordinates

Since a network coordinate system assigns coordinates based on the measured latency between the hosts, one should expect to see clusters of peers if one plots the coordinates that correspond to peers on different continents. Theses clusters should be separated by gaps, the oceans. In figure 12.15(a), we plotted the network coordinates version 1 (without the height) of the German and the Australian peers. There is a strong overlap between them, no clear separation and no gap. This is the same for other pairs of countries plotted together. In figure 12.15(b), the coordinates of peers of all countries are plotted. It is not possible to distinguish between countries or even continents. Geographical distances are not at all represented.

These findings are in contradiction to Ledlie at al. [65] who claims that embedding the Internet running on a globe (the Earth) into an Euclidean space works fine, due to the fact that traffic between Europe and Asia is routed via the US. They state that the peers of different continents (Asia, Europe, and North America) cluster together in the network coordinate space, which is in clear disaccord to our findings. In their technical report [66], they show (Figure 11) snapshots of the coordinates of the peers they run on PlanetLab. Three clusters that represent the continents are distinguishable. We believe that such results can be obtained on PlanetLab but with peers that are connected via ADSL.

12.4.4 Stability of the network coordinates

The stability of the coordinates is extremely important if they trigger events (e.g. a download) that may take several hours. Assuming there is no change in the network, a coordinate system is stable if the coordinates of the nodes in the system are stable. In order to analyze the stability, we decided that a temporal resolution of 8 hours (3 crawls a day) is not enough and dumped the coordinates of 2 peers (running on 2



(a) Germany and Australia: overlap, no separation, no gap.

(b) All countries: no distinction of continents possible.

Figure 12.15: AZUREUS network coordinates version 1

machines in the same LAN) at Eurécom, 2 peers (running on 2 machines in the same LAN) at the University of Mannheim and one connected via a France Télécom ADSL line every 5 minutes for 9 days, starting on February 25, 2008. From this data set, we calculated the speed at which the coordinates change. x_t is the coordinate of peer x at time t, $x_{t+\delta}, \delta = 300$ seconds is the position of the same peer x on the next crawl 5 minutes later. The speed of change is $\frac{x_{t+\delta}-x_t}{\delta}$. The unity of the speed of coordinate change is $msecs/sec$.

For version 1, we observed an average speed of 0.48 $msecs/sec$ (0.52 standard deviation, 6.9 max). Version 2 shows an improvement of factor 2 with an observed average speed of only 0.19 $msecs/sec$ on average (0.17 standard deviation, 1.1 max). Ledlie at al. (see [65], Figure 7) however claim an improvement in stability of 4 orders of magnitude.

12.4.5 Which peer to choose?

The classical use of an Internet coordinate system is to choose the peers from which to download from. To check if the coordinate system implemented in AZUREUS fulfills that request, we set up a very simple experiment. We dumped the coordinates of 2 peers (running on 2 machines in the same LAN) at Eurécom, 2 peers (running on 2 machines in the same LAN) at the University of Mannheim, and one connected via a France Télécom ADSL line every 5 minutes for 9 days, starting on February 25, 2008. Using the coordinates, the peers located in Mannheim, respectively Eurécom should be able to choose the other peer in the same LAN.

In the following, we computed the Vivaldi distances between those peers (Table 12.3). When using network coordinates, the distance values for the different pairs of peers make these pairs practically indistinguishable, or as in case of version 2, make two peers in Eurécom / Mannheim look closer to each other than peers that are adjacent.

peers	ICMP	version 1			version 2		
	ping	avg.	st.dev.	max	avg.	st.dev.	max
ma 1 / ma 2	0	133	117	1,208	121	56	336
eur 1 / eur 2	0	182	204	2,606	140	72	414
ma 1 / eur 2	39	158	120	786	95	45	302
ma 2 / eur 1	39	144	171	2,563	167	71	390
eur 1 / adsl	70	177	126	1,066	108	37	241
ma 1 / adsl	70	179	121	981	106	37	269

Table 12.3: Vivaldi distances between two peers. The first column indicates the location of the two peers, the second column indicates the ICMP ping times between them.

12.5 Conclusion

We have studied the network coordinate system currently implemented in AZUREUS and evaluated its possibilities and limitations. We saw that the latencies estimated using network coordinates exhibit a high correlation with the round trip times at application layer and to a lesser degree with the round trip times at network layer. The round trip time is composed of three elements: propagation delay, transmission delay, and queuing delay. As many peers are connected to the Internet via ADSL, the queuing delay of the ADSL access link can dominate the round trip time and hide the contribution of the geographical distance completely, which is reflected in the propagation delay. Extremely long round trip times of several seconds are a strong indication for heavily loaded ADSL links and not for a huge distance between those peers. Due to this fact, groups of peers in geographical proximity, e.g. countries or ISPs, are not reflected in the coordinate space. As a result, network coordinates cannot be used to reliably select "nearby" peers that are in the same ISP or country. Instead, we suggest to select "nearby" peers one could use reverse DNS lookup on the peers IP addresses, which AZUREUS already uses today to determine the country of origin of the peers. However, network coordinates are still very useful for peer selection whenever the round trip time has an impact on the performance. As it is, for instance, the case in query routing and when downloading content using TCP.

Chapter 13
Conclusion of Part II

"One of the most feared expressions in modern times is "The computer is down"."
– Norman Augustine –

In this part of the thesis, we presented measurement techniques to learn about participants and content in peer-to-peer networks.

First, we implemented our crawler *Blizzard* that allows for crawling millions of users in a DHT in the order of minutes. In KAD, the first million users are detected after 10 seconds only. To our knowledge, today, there exists no other crawler that is able to crawl an entire large-scale DHT.

We observed that the peers seen first during the very first crawl had much higher mean session durations and smaller inter-session times than peers seen later for the first time. As already suggested by Stutzbach [123], this fact can be exploited to find "more stable" peers without knowing anything about the history of the peers: One simply "crawls" KAD *once* and selects the peers that are online in that instant. We also saw that the availability of peers in China is much lower than the one of peers in Europe.

Crawling these different networks, KAD (Section 9) and AZUREUS (Section 12), we can make some general remarks. Each network is predominant in one region of the world, KAD in Europe, and China and Azureus in North America. The median session times of the peer is in the order of hours and most users are on-line in the evening of their local time zone. A big fraction of the users is connected to the Internet with a classical ADSL line, the ten largest ISPs for the users of both peer-to-peer networks are ADSL providers.

The major difference in the implementation is the two level publishing scheme in KAD (cf. Section 8.2) that allows end-users for keyword searching against the one level publishing scheme in AZUREUS that does not allow for keyword searching by end-users. In AZUREUS it is only possible to search for torrents that have already been downloaded from a website: the DHT is a fall-back mechanism.

Second, we developed the content spy *Mistral* that allows for getting insight on what content is shared in a DHT. We did extensive measurement in the KAD network (Section 10).

In today's implementation of KAD, a source key that points to the peer holding the content will expire 5 hours after it has been published. On the other hand, we observed that the median session length of peers is 155 minutes. Also, less than 40% of the peers have a session length of 5 hours or more (cf. figure 9.12). This means that, in more than 60% of the cases, the peer that publishes a source key will leave KAD before the reference to that file will expire. As a result, many references with an expiration time of 5 hours to sources in KAD will be *stale*, resulting in unsuccessful attempts to download that file. An improvement of the current implementation could be to first publish a source key with an expiration time much smaller than 5 hours and to increase the expiration time progressively as the uptime of the peer that owns the file increases, exploiting the fact that session times are Weibull distributed.

The results presented in this part can be used to improve the performance of the implementation of KAD in various ways. We measured (Section 10) that, due to publishing, the total traffic in KAD is about 100 times higher in volume (bytes) than the total search traffic. Carra et al. [18] have shown how to exploit the fact that session times are Weibull distributed in order to reduce the publish traffic by one order of magnitude.

The content spy *Mistral* relies on the so-called *Sybil* attack. This leads us to study in detail how vulnerable KAD is to different kind of attacks. We showed that an *Eclipse* attack against some content in KAD is easily feasible even with scarce resources. We also presented results of a *content pollution attack* we ran against the peer-to-peer network of the Storm worm.

Furthermore, we studied in detail the content management process implemented in aMule and we show that the design approach has a strong impact on the overall lookup latency in case of content retrieval (Section 11). We made proposals on how to speed up the overall look up latency without decreasing the efficiency.

The engineers who implemented Kademlia in aMule and eMule had to make some important design choices: the number of times a publications is replicated, the size of the zone in which a publication might be spread around its hash (tolerance zone), the degree of parallelism for the look up, the time to wait for closer peers before launching the content retrieval, etc. In general, one can say that most parameters are well chosen. The resulting algorithm allows for fuzziness in the publication and the look up process. However, due to the high number of replications, the content is always found without the need that the peers on which some content is published have to hand over the content to another peer before they leave. The tolerance zone has been fixed to 8 bit, this might have been a good choice for the early days of the network, when an 8-bit zone did contain only a few hundred peers. The developers probably could not anticipate the future success of the network. An 8-bit zone of today's KAD network contains around 8,000 peers. All those peers are qualified to receive a publication for a keyword that shares the first 8 bit with them. The con-

sequence is that a keyword being published at the border of the tolerance zone will most probably never be found and is therefore wasted.

Chapter 14
Summary

"Computer science only indicates the retrospective omnipotence of our technologies. In other words, an infinite capacity to process data (but only data – i.e. the already given) and in no sense a new vision. With that science, we are entering an era of exhaustivity, which is also an era of exhaustion."

– Jean Baudrillard –

Peer-to-peer systems represent a new way of offering services at the edges of the Internet by users and for users. Since more and more users are connected to the Internet, the scalability issued gains importance. Peer-to-peer systems are worth considering for new Internet services to be developed since in a peer-to-peer network a new peer does not only stand for more load but also for more resources. However, the distributed nature of peer-to-peer networks and especially the non-existence of a centralized management requires self-organizing algorithms that are able to smoothly scale to millions of users and use the resources of a very large number of machines connected through the Internet efficiently. Today, peer-to-peer networks have been successfully deployed to millions of users and offer services such as file sharing, audio and video streaming, telephony, backup, and games.

As detailed in the introduction of this thesis, peer-to-peer networks can be separated in two main classes: *unstructured* and *structured* overlay networks.

In the first part of this thesis, we presented a set of algorithms for the dynamic maintaining of a distributed overlay network matching with the Delaunay triangulation of the entities. We showed how to augment this triangulation by additional shortcuts which reflect the underlaying Internet to build a small world overlay that reduces both, average number of hops and delay. We moreover presented a dynamic algorithm to cluster nodes in a totally distributed way.

With the help of the developed algorithm, it is possible to create a virtual world, a networked virtual environment, relying on an unstructured peer-to-peer overlay network. A networked virtual environment is an application that fits nicely to the design of an unstructured overlay network. Since every participant in a virtual world is basically interested in his direct neighbors, there is no need to efficiently find some specific content on the other side of the virtual world. Such a system could solve the scalability issue encountered by the huge success story of massive multiplayer online games such as World of Warcraft that has attracted more than six million subscribers within two years.

Structured peer-to-peer networks by means of Distributed Hash Tables present interesting properties (scalability to a large population of users, fast look-up of resources) for content distribution and content location. In a file-sharing network, the users want to know on which machine a specific file is stored, a DHT does exactly provide the answer to that question.

In the second part of the thesis, we present an extensive measurement study of such a structured peer-to-peer file-sharing network, namely KAD , the largest currently deployed DHT that is used by millions of users. We developed our peer crawler *Blizzard* that allows to discover and contact all peers in the network in the order of some minutes only. To our knowledge, no crawler with a similar performance has been presented up to now. The crawler is not only fast but also flexible, it can be used in any DHT, e.g. we used it to crawl the networks of AZUREUS and the *Storm Worm*.

The results of the crawls of the KAD network we performed for over a year lead to a number of interesting findings. For the first time, the total number of peers online could be measured precisely. To our big surprise, some of the peers stayed connected during the whole observation period. This high availability by home users connected via ADSL is very amazing. The session times of the peers are heavy tailed following a Weibull distribution. Most users seem to be very satisfied by the experience KAD offers, they use KAD every day for many hours and come back over and over again. 50% of the users have been using KAD for six months and more. We discovered that the identifiers of the peers in the DHT are not necessarily persistent as was assumed up to now. Nevertheless, the most important metrics such as session times and inter-session times are not affected by the non-persistent identifiers.

Beside the peer behavior, we also analyzed the content published and searched in KAD. Therefore, we developed *Mistral*, a content spy, that allows us to gain an overview of the content published and searched in KAD. Our observations show that the publication process in KAD is responsible for more than 90% of the total network traffic. Moreover, we note that the load is highly unbalanced between the peers. We made proposals how to reduce the overhead and as a consequence, improve the overall system performance.

While developing *Mistral*, we discovered how easily peer-to-peer systems like KAD can be attacked. In fact, *Mistral* relies on a so-called *Sybil attack*. Moreover, we implemented an *Eclipse attack* that allows to hide content from KAD users. Such attacks

are feasible even with scarce resources. KAD can also be misemployed to launch a distributed DDoS attack, even against machines that do not participate in KAD. We described mechanisms to prevent such attacks in a peer-to-peer system.

In this thesis on peer-to-peer networks, we have been looking at network virtual environments which are an application that profits of unstructured peer-to-peer networks. To build such virtual worlds in a peer-to-peer way, we have proposed a set of distributed and dynamic algorithms. Furthermore, we presented measurement techniques and extensive measurement results on a real-world, large-scale peer-to-peer file sharing network, namely KAD, which relies on a DHT, a structured peer-to-peer network.

Bibliography

[1] A-Mule. http://www.amule.org/.

[2] K. Aberer, F. Klemm, M. Rajman, and J. Wu. An architecture for peer-to-peer information retrieval. In *Workshop on Peer-to-Peer Information Retrieval*, 2004.

[3] F. Araujo and L. Rodrigues. GeoPeer: A Location-Aware Peer-to-Peer System. Technical report, Faculdade de Ciências da Universidade de Lisboa, Portugal, 2001.

[4] E. Athanasopoulos, K. G. Anagnostakis, and E. P. Markatos. Misusing Unstructured P2P Systems to Perform DoS Attacks: The Network that Never Forgets. In *Proceedings of ACNS*, June 2006.

[5] F. Aurenhammer and R. Klein. *Handbook of Computational Geometry*, chapter 18, pages 201–290. Elsevier Science Publishers, 2000.

[6] Azureus. http://azureus.sourceforge.net/.

[7] S. Benford, J. Bowers, L. E. Fahlén, and C. Greenhalgh. Managing Mutual Awareness in Collaborative Virtual Environments. In *ACM Symposium on Virtual Reality Software and Technology (VRST)*, 1994.

[8] R. Bhagwan, S. Savage, and G. Voelker. Understanding availability. In *Proceedings of the 2^{nd} International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 256–267, 2003.

[9] C. Blake and R. Rodrigues. High Availability, Scalable Storage, Dynamic Peer Networks: Pick two. In *HotOs*, 2003.

[10] J. D. Boissonnat and M. Teillaud. The Hierarchical Representation of Objects: the Delaunay Tree. In *ACM Symposium on Computational Geometry*, 1996.

[11] P. Bose and P. Morin. Online Routing in Triangulations. In *Proceedings of the 10^{th} International Symposium on Algorithms and Computation (ISAAC)*, pages 113–122, 1999.

[12] O. Boxma and B. Zwart. Tails in scheduling. *Performance Evaluation Review*, 34(4), 2007.

[13] E. Bradner and G. Mark. Why distance matters: effects on cooperation, persuasion and deception. In *Proceedings of the conference on Computer supported cooperative work (CSCW)*, pages 226–235, 2002.

[14] F. Bustamante and Y. Qiao. Designing Less-structured P2P Systems for the Expected High Churn. *IEEE/ACM Transactions on Networking (TON)*, 16(3), June 2008.

[15] K. Calvert, M. Doar, and E. W. Zegura. Modeling internet topology. *IEEE Communications Magazine*, 35(6):160–163, June 1997.

[16] K. Calvert, J. Eagan, S. Merugu, A. Namjoshi, J. Stasko, and E. Zegura. Extending and Enhancing GT-ITM. In *Proceedings of the SIGCOMM workshop on Models, methods and tools for reproducible network research*, pages 23–27, 2003.

[17] J. Calvin, A. Dickens, B. Gaines, P. Metzger, D. Miller, and D. Owen. The SIMNET Virtual World Architecture. In *Annual International Symposium Virtual Reality IEEE*, pages 450–455, Sept. 1993.

[18] D. Carra and E. Biersack. Building a Reliable P2P System Out of Unreliable P2P Clients: The Case of KAD. In *Proceedings of CoNEXT*, Dec. 2007.

[19] E. Castillo. *Extreme Value Theory in Engineering*. Academic Press, Inc., 1988.

[20] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. Wallach. Secure routing for structured peer-to-peer overlay networks. In *Proceedings of OSDI*, Dec. 2002.

[21] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Topology-aware routing in structure Peer-to-peer overlay network. In *International Workshop on Future Directions in Distributed Computing (FuDiCo)*, Bertinoro, Italy, June 2002.

[22] J. Chu, K. Labonte, and B. N. Levine. Availability and locality measurements of peer-to-peer file systems. In *Proceedings of SPIE*, July 2002.

[23] A. T. Clements, D. R. K. Ports, and D. R. Karger. Arpeggio: Metadata searching and content sharing with chord. In *International Workshop on Peer-To-Peer Systems*, 2005.

[24] E. Cooke, F. Jahanian, and D. McPherson. The zombie roundup: Understanding, detecting, and disrupting botnets. In *Workshop on Steps to Reducing Unwanted Traffic on theInternet (SRUTI)*, pages 39–44, June 2005.

[25] M. Costa, M. Castro, A. Rowstron, and P. Key. Pic: Practical internet coordinates for distance estimation. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS)*, pages 178–187, 2004.

[26] R. Cox, F. Dabek, F. Kaashoek, J. Li, and R. Morris. Vivaldi: A Decentralized Network Coordinate System. In *Proceedings ACM SIGCOMM*, Aug. 2004.

[27] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris. Designing a DHT for low latency and high throughput. In *NSDI*, 2004.

[28] G. Danezis, C. Lesniewski-Laas, M. Kaashoek, and R. Anderson. Sybil-resistant DHT routing. In *European Symposium On Research In Computer Security*, 2005.

[29] B. Delaunay. Sur la sphère vide. A la mémoire de Georges Voronoi. *Izv. Akad. Nauk SSSR, Otdelenie Matematicheskih i Estestvennyh Nauk*, 7:793–800, 1934.

[30] O. Devillers. On Deletion in Delaunay Triangulations. In *Proceedings of the Fifteenth Annual Symposium on Computational Geometry (SGC)*, pages 181–188, 1999.

[31] J. Dinger and H. Hartenstein. Defending the Sybil Attack in P2P Networks: Taxonomy, Challenges, and a Proposal for Self-Registration. In *Proceedings of the First International Conference on Availability, Reliability and Security (ARES)*, pages 756–763, 2006.

[32] M. Dischinger, A. Haeberlen, K. P. Gummadi, and S. Saroiu. Characterizing residential broadband networks. In *Proceedings of the Internet Measurement Conference (IMC)*, Oct. 2007.

[33] J. R. Douceur. The Sybil attack. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, LNCS, pages 251–260, Mar. 2002.

[34] E-Mule. http://www.emule-project.net/.

[35] H. Edelsbrunner and N. R. Shah. Incremental topological flipping works for regular triangulations. *Algorithmica*, 15:223–241, 1996.

[36] K. El-Defrawy, M. Gjoka, and A. Markopoulou. BotTorrent: Misusing BitTorrent to Launch DDoS Attack. In *Proceedings of USENIX SRUTI*, June 2007.

[37] C. Eldershaw and M. Hegland. Cluster Analysis using Triangulation. In *Computational Techniques and Applications*, 1997.

[38] V. Estivill-Castro and I. Lee. AUTOCLUST: Automatic Clustering via Boundary Extraction for Mining Massive Point-data Sets. In *Proceedings of the 5th International Conference on GeoComputation University of Greenwich*, pages 82–89, Aug. 2000.

[39] V. Estivill-Castro, I. Lee, and A. T. Murray. Criteria on proximity graphs for boundary extraction and spatial clustering. In *Proceedings of the 5th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*, pages 348–357, 2001.

[40] J. Falkner, M. Piatek, J. P. John, A. Krishnamurthy, and T. Anderson. Profiling a Million User DHT. In *Proceedings of the Internet Measurement Conference (IMC)*, Oct. 2007.

[41] F. L. Fessant, S. B. Handurukande, A.-M. Kermarrec, and L. Massoulié. Clustering in peer-to-peer file sharing workloads. In *The 3rd International Workshop on Peer-to-Peer Systems (IPTPS)*, Oct. 2004.

[42] S. Fiedler, M. Wallner, and M. Weber. A communication architecture for massive multiplayer games. In *Workshop on Network and systems support for Games (NetGames)*, 2002.

[43] C. Goldie and C. Klueppelberg. *Subexponential Distributions*, pages 435–455. Birkhauser, Basel, 1997.

[44] P. J. Green and R. R. Sibson. Computing dirichlet tessellations in the plane. *Computer Journal*, 21:168–173, 1978.

[45] L. Grossman. The Worm That Roared. Internet: http://www.time.com/time/magazine/, Sept. 2007.

[46] L. Guibas, D. Knuth, and M. Sharir. Randomized Incremental Constructions of Delaunay and Voronoi Diagrams. *Algorithmica*, 7:381–413, 1992.

[47] 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 SIGCOMM*, 2003.

[48] K. P. Gummadi, S. Saroiu, and S. D. Gribble. King: Estimating Latency between Arbitrary Internet End Hosts. In *Proceedings of 2nd Internet Measurement Workshop*, Marseille, France, Nov. 2002.

[49] I. Gupta, A.-M. Kermarrec, and A. J. Ganesh. Efficient epidemic-style protocols for reliable and scalable multicast. In *Proceedings of the 21st Symposium on Reliable Distributed Systems (SRDS)*, page 180, 2002.

[50] M. Gutowski. Lévy flights as an underlying mechanism for global optimization algorithms. *ArXiv Mathematical Physics e-prints*, June 2001.

[51] T. Holz, M. Steiner, F. Dahl, E. W. Biersack, and F. Freiling. Measurements and Mitigation of Peer-to-Peer-based Botnets: A Case Study on Storm Worm. In *First Usenix Workshop on Large-scale Exploits and Emergent Threats (LEET)*, San Francisco, CA, USA, Apr. 2008.

[52] Honeynet Project. Know your Enemy: Tracking Botnets, Mar. 2005. http://www.honeynet.org/papers/bots.

[53] S.-Y. Hu, J.-F. Chen, and T.-H. Chen. VON: a scalable peer-to-peer network for virtual environments. *IEEE Network*, 20(4):22–31, 2006.

[54] Y. hua Chu, S. G. Rao, S. Seshan, and H. Zhang. A Case for End System Multicast. *IEEE Journal on Selected Areas in Communication, Special Issue on Networking Support for Multicast*, 20, Oct. 2002.

[55] K. Hui, J. Lui, and D. Yau. Small world overlay p2p networks. In *Proceedings of the Twelfth International Workshop on Quality of Service (IWQOS)*, pages 201–210, 2004.

[56] M. Izal, G. Urvoy-Keller, E. Biersack, P. Felber, A. Al Hamra, and L. Garces-Erice. Dissecting BitTorrent: Five Months in a Torrent's Lifetime. In *Proceedings of the Passive and Active Measurement Conference*, Apr. 2004.

[57] KadC. http://kadc.sourceforge.net/.

[58] I.-S. Kang, T. wan Kim, and K.-J. Li. A spatial data mining method by Delaunay triangulation. In *Proceedings of the 5th international workshop on Advances in geographic information systems (GIS)*, pages 35–39, 1997.

[59] J. Keller and G. Simon. SOLIPSIS: A Massively Multi-Participant Virtual World. In *Proceedings of the International Conference on Parallel and Distributed Techniques and Applications*, 2003.

[60] J. Kleinberg. The small-world phenomenon: an algorithm perspective. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing (STOC)*, pages 163–170, 2000.

[61] F. Klemm, A. Datta, and K. Aberer. A Query-Adaptive Partial Distributed Hash Table for Peer-to-Peer Systems. In *Current Trends in Database Technology - EDBT Workshops*, pages 506–515, 2004.

[62] B. Knutsson, H. Lu, W. Xu, and B. Hopkins. Peer-to-Peer Support for Massively Multiplayer Games. In *Proceedings of INFOCOM*, Mar. 2004.

[63] B. Krebs. Storm worm dwarfs world's top supercomputers. Internet: `http://blog.washingtonpost.com/securityfix/`, Aug. 2007.

[64] K. Kutzner and T. Fuhrmann. Measuring Large Overlay Networks - The Overnet Example. In *Proceedings of the 14th KiVS*, Feb. 2005.

[65] J. Ledlie, P. Gardner, and M. Seltzer. Network Coordinates in the Wild. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design & Implementation*, pages 299–311, 2007.

[66] J. Ledlie, P. Gardner, and M. Seltzer. Network Coordinates in the Wild. Technical report, University of Harvard, Cambridge, MA, US, 2007.

[67] K. A. Lehmann and M. Kaufmann. *Random Graphs, Small-Worlds and Scale-Free Networks*, volume 3485 of *LNCS*, chapter 6, pages 57–76. Springer, 2005.

[68] E. Léty, T. Turletti, and F. Bacceli. Score: a scalable communication protocol for large-scale virtual environments. *IEEE Transactions on Networking (TON)*, 12(2):247–260, 2004.

[69] P. Lévy. *Théorie de l'Addition des Variables Aléatoires*. Gauthier-Villiers, Paris, 1937.

[70] J. Li, J. Stribling, R. Morris, M. Kaashoek, and T. Gil. A performance vs. cost framework for evaluating DHT design tradeoffs under churn. In *Proceedings of INFOCOM*, 2005.

[71] M. Li, W.-C. Lee, and A. Sivasubramaniam. Semantic small world: An overlay network for peer-to-peer search. In *Proceedings of the 12th IEEE International Conference on Network Protocols (ICNP)*, pages 228–238, 2004.

[72] X. Li, G. Calinescu, and P.-J. Wan. Distributed Construction of a Planar Spanner and Routing for Ad Hoc Wireless Networks. In *Proceedings of INFOCOM*, 2002.

[73] J. Liebeherr, M. Nahas, and S. Weisheng. Application-layer multicasting with Delaunay triangulation overlays. *IEEE Journal on Selected Areas in Communications*, 20(8):1472–1488, Oct. 2003.

[74] Y. Liu, Z. Zhuang, L. Xiao, and L. M. Ni. A distributed approach to solving overlay mismatching problem. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS)*, pages 132–139, 2004.

[75] B. B. Mandelbrot. *Fractals : form, chance, and dimension*. San Francisco : W.H. Freeman, 1977.

[76] Maxmind. http://www.maxmind.com/.

[77] P. Maymounkov and D. Mazieres. Kademlia: A Peer-to-peer information system based on the XOR metric. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 53–65, Mar. 2002.

[78] S. Merugu, S. Srinivasan, and E. Zegura. Adding structure to unstructured peer-to-peer networks: The role of overlay topology. In *Proceedings of NGC*, volume 2816, pages 83–94, Sept. 2003.

[79] J. Mickens and B. Noble. Exploiting availability prediction in distributed systems. In *Proceedings of NSDI*, 2006.

[80] S. Milgram. The small world problem. *Psychology Today*, 2:60–67, May 1967.

[81] I. Mitrani. *Probabilistic Modelling*. Cambridge University Press, 1998.

[82] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge Press, 2005.

[83] M. A. Mostafavia, C. Gold, and M. Dakowicz. Delete and insert operations in voronoi/delaunay methods and applications. In *Computers & Geosciences*, volume 29, pages 523–530, 2003.

[84] N. Naoumov and K. Ross. Exploiting P2P Systems for DDoS Attacks. In *International Workshop on Peer-to-Peer Information Management*, May 2006.

[85] J. Naughton. In millions of windows, the perfect storm is gathering. http://observer.guardian.co.uk/, Oct. 2007.

[86] M. E. J. Newman, D. J. Watts, and S. H. Strogatz. Random graph models of social networks. In *PNAS*, volume 99, pages 2566–2572, 2002.

[87] T. E. Ng and H. Zhang. A network positioning system for the internet. In *USENIX Annual Technical Conference*, pages 141–154, 2004.

[88] A. Okabe, B. Boots, and K. Sugihara. *Spatial Tesselations: Concepts and Applications of Voronoi Diagrams*. John Wiley and Sons, 2000.

[89] Overnet. http://www.overnet.org/.

[90] M. Pias, J. Crowcroft, S. Wilbur, and T. H. S. Bhatti. Lighthouses for scalable distributed location. In *Proceedings of 2nd International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 278–291, Berkeley, CA, USA, Feb. 2003.

[91] M. Pietrzyk, G. Urvoy-Keller, and J.-L. Costeux. Digging into KAD Users' Shared Folders. In *Poster of SIGCOMM*, 2008.

[92] I. Podnar, M. Rajman, T. Luu, F. Klemm, and K. Aberer. Beyond term indexing: A p2p framework forweb information retrieval. *Informatica*, 30:153–161, 2006.

[93] J. Postel. Transmission Control Protocol – Protocol Specification. Request for Comments (Standard) RFC 793, Information Sciences Institute, USC, Sept. 1981.

[94] Prolexic. Prolexic Distributed Denial of Service Attack Alert, May 2007. http://www.prolexic.com/news/20070514-alert.php.

[95] J. Purbrick and C. Greenhalgh. Extending Locales: Awareness Management in MASSIVE-3. In *IEEE Virtual Reality Conference (VR)*, 2000.

[96] Y. Qiao and F. E. Bustamante. Structured and Unstructured Overlays Under the Microscope - A Measurement-based View of Two P2P Systems That People Use. In *Proceedings of the USENIX Annual Technical Conference*, 2006.

[97] M. A. Rajab, J. Zarfoss, F. Monrose, and A. Terzis. A multifaceted approach to understanding the botnet phenomenon. In *Proceedings of the 6th Internet Measurement Conference*, 2006.

[98] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of SIGCOMM*, 2001.

[99] H. Rowaihy, W. Enck, P. McDaniel, and T. La Porta. Limiting Sybil Attacks in Structured P2P Networks. In *Proceedings of INFOCOM)*, pages 2596–2600, 2007.

[100] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale Peer-to-peer systems. In *Proceedings of Middleware*, Heidelberg, Germany, Nov. 2001.

[101] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking (MMCN)*, Jan. 2002.

[102] S. Sen and J. Wang. Analyzing peer-to-peer traffic across large networks. *ACM/IEEE Transactions on Networking (ToN)*, 2004.

[103] R. Siebes. pNear: combining Content Clustering and Distributed Hash Tables. *In Proceedings of the IEEE - p2pkm*, 2005.

[104] A. Singh, M. Castro, P. Druschel, and A. Rowstron. Defending against eclipse attacks on overlay networks. In *SIGOPS*, 2004.

[105] A. Singh et al. Eclipse Attacks on Overlay Networks: Threats and Defenses. In *Proceedings of INFOCOM*, Apr. 2006.

[106] E. Sit and R. Morris. Security Considerations for Peer-to-peer Distributed Hash Tables. In *Proceedings of IPTPS*, Cambridge, MA, Mar. 2002.

[107] K. Sripanidkulchai. The popularity of gnutella queries and its implications on scalability. In *Proceedings of O'Reilly's OpenP2P*, 2001.

[108] M. Steiner and E. Biersack. Shortcuts in a Virtual World. In *Proceedings of CoNext*, 2006.

[109] M. Steiner and E. W. Biersack. DDC: A Dynamic and Distributed Clustering Algorithm for Networked Virtual Environments based on P2P networks. In *Proceedings of CoNEXT (Poster)*, Toulouse, France, Oct. 2005.

[110] M. Steiner and E. W. Biersack. A fully distributed peer to peer structure based on 3D Delaunay Triangulation. In *Proceedings of Algotel - Septièmes Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications*, pages 93–96, May 2005.

[111] M. Steiner and E. W. Biersack. DDC: A Dynamic and Distributed Clustering Algorithm for Networked Virtual Environments based on P2P networks. In *Proceedings of the 9^{th} IEEE Global Internet Symposium in conjunction with IEEE Infocom*, Barcelona, Spain, Apr. 2006.

[112] M. Steiner, E. W. Biersack, and T. En-Najjary. Actively Monitoring Peers in Kad. In *Proceedings of the 6^{th} International Workshop on Peer-to-Peer Systems (IPTPS)*, 2007.

[113] M. Steiner, E. W. Biersack, and T. En-Najjary. Exploiting KAD: Possible Uses and Misuses. *Computer Communication Review*, 37(5), Oct. 2007.

[114] M. Steiner, E. W. Biersack, and T. En-Najjary. Long Term Study of Peer Behavior in the KAD DHT. *Accepted for Publication in IEEE/ACM Transactions on Networking*, 2008.

[115] M. Steiner, D. Carra, and E. W. Biersack. Faster content access in kad. In *Proceedings of the 8^{th} IEEE Conference on Peer-to-Peer Computing (P2P)*, Aachen, Germany, Sept. 2008.

[116] M. Steiner, W. Effelsberg, T. En-Najjary, and E. W. Biersack. Load Reduction in the KAD Peer-to-Peer System. In *Fifth International Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P 2007)*, 2007.

[117] M. Steiner, T. En-Najjary, and E. W. Biersack. A Global View of KAD. In *Proceedings of the Internet Measurement Conference (IMC)*, 2007.

[118] N. Stephenson. *Snow Crash*. Bantam Spectra Book, 1992.

[119] J. Stewart. Storm worm DDoS attack. Internet: `http://www.secureworks.com/research/threats/storm-worm`, 2007.

[120] 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 SIGCOMM*, pages 149–160, 2001.

[121] D. Stutzbach and R. Rejaie. Evaluating the accuracy of captured snapshots by peer-to-peer crawlers. In *Proceedings of PAM*, volume 3431, pages 353–357, Jan. 2005.

[122] D. Stutzbach and R. Rejaie. Improving Lookup Performance over a Widely-Deployed DHT. In *Proceedings of INFOCOM*, Apr. 2006.

[123] D. Stutzbach and R. Rejaie. Understanding churn in peer-to-peer networks. In *Proceedings of the Internet Measurement Conference (IMC)*, Oct. 2006.

[124] T. Suel, C. Mathur, J.-W. Wu, J. Zhang, A. Delis, M. Kharrazi, X. Long, and K. Shanmugasundaram. ODISSEA: A Peer-to-Peer Architecture for Scalable Web Search and Information Retrieval. Technical Report TR-CIS-2003-01, Department of Computer and Information Science, Polytechnic University, Brooklyn, NY, June 2003.

[125] M. Szymaniak, G. Pierre, and M. van Steen. Latency-Driven Replica Placement. In *Proceedings of the Symposium on Applications and the Internet (SAINT)*, pages 399–405, 2005.

[126] C. Tang, M. J. Buco, R. N. Chang, S. Dwarkadas, L. Z. Luan, E. So, and C. Ward. Low traffic overlay networks with large routing tables. In *Proceedings of SIGMETRICS*, pages 14–25, 2005.

[127] L. Tang and M. Crovella. Virtual Landmarks for the Internet. In *Proceedings of the Internet Measurement Conference (IMC)*, Oct. 2003.

[128] The Internet Movie Database. `http://www.imdb.com/`.

[129] J. Tian and Y. Dai. Understanding the Dynamic of Peer-to-Peer Systems. In *Proceedings of the 6^{th} International Workshop on Peer-to-Peer Systems (IPTPS)*, 2007.

[130] F. D. Tran, M. Deslaugiersa, A. Gerodolle, L. Hazard, and N. Rivierre. An open middleware for large-scale networked virtual environments. In *IEEE Virtual Reality Conference (VR)*, 2002.

[131] K. Tutschku. A measurement-based traffic profile of the edonkey filesharing service. In *5th Passive and Active Measurement Workshop (PAM2004)*, Apr. 2004.

[132] G. Voronoï. Nouvelles applications des paramètres continus à la théorie des formes quadratiques. Deuxième mémoire: Recherches sur les parallellotòres primitifs. *Journal für die Reine and Angewandte Mathematik*, 134:198–287, 1908.

[133] S. Voulgaris and M. van Steen. Epidemic-style management of semantic overlays for content-based searching. In J. C. Cunha and P. D. Medeiros, editors, *Proceedings of the 11^{th} International Euro-Par Conference*, volume 3648, page 1143, Aug. 2005.

[134] R. Waters and W. Yerazunis. Diamond park and spline: A social virtual reality system with 3d animation, spoken interaction, and runtime modifiability. *Presence: Teleoperators and Virtual Environments*, 6(4):461–480, 1997.

[135] D. Watson. Computing the n-dimensional Delaunay triangulation with application to Voronoi polotypes. *The Computer Journal*, 24(2):167–172, 1981.

[136] D. Watts and S. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):409–410, 1998.

[137] D. J. Watts. *Small worlds: the dynamics of networks between order and randomness*. Princeton University Press, Princeton, NJ, USA, 1999.

[138] N. Yee. The demographics, motivations and derived experiences of users of massively-multiuser online graphical environments. *Presence: Teleoperators and Virtual Environments*, 15:309–329, 2006.

[139] H. Yu, M. Kaminsky, P. B. Gibbons, and A. Flaxman. SybilGuard: Defending Against Sybil Attacks via Social Networks. In *Proceedings of SIGCOMM*, 2006.

[140] B. Y. Zhao, J. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, Computer Science Division, University of California, Berkeley, Apr. 2001.

[141] S. Zhou, W. Cai, B.-S. Lee, and S. J. Turner. Time-space consistency in large-scale distributed virtual environments. *ACM Transactions on Modeling and Computer Simulation*, 14(1):31–47, 2004.

[142] M. Zyda. Npsnet-iv: Inserting the human into the networked synthetic environment. In *ACM SIGGRAPH*, 1997.

APPENDIX

APPENDIX A

Experiences in Data Management

Problem Statement Crawling KAD every 5 minutes for 6 months was not only a challenge from the crawling point of view but also for the data management. Remember that we run two crawlers in parallel at two different locations, every single crawl discovers about 20,000 peers of which about 10,000 respond. Every of the more than 100,000 crawls is saved to a file with an average size of 1.2 MB. In total this accounts for 125 GB of data.

Extracting the behavior of a single client, e.g. its session times, the first join, or the last leave, yields in 100,000 *greps*. Having several 100,000 KAD IDs this is not feasible.

For the full crawl that we run for more than one year once a day the dataset is not smaller. Every crawl discovers about 4 million peers of which 1.5 million respond. This accounts for a file of 230 MB per day. In total this accounts for about 100 GB of data.

Database The solution we found was to use a standard open source mysql[1] database, however specially configured for our needs. No need to execute a big number of queries in parallel, therefore all the resources can be attributed to one single query at a time. We optimized all the parameters in a way, such that the performance is optimized for a single reading request returning a huge (containing all the data of the data base) dataset.

In order to reduce the size of the data in the data base the data types have to be chosen very carefully. Using the `INET_ATON` function of mysql we converted the IP addressesto Integer values. With the `unhex` function we converted the first 64 out of 128 bits of the KAD ID to a mysql bit field. The first 64 bits are enough since our measurements confirmed that benign peers have never more than 32 bits in common. For the full crawl we used the `date` data type and for the zone crawl the `datetime` that has a time granularity of one second.

[1] www.mysql.com

```
mysql> create table newnewpivot
(ip int unsigned, port smallint unsigned, hash bit(64));

+-------+----------------------+
| Field | Type                 |
+-------+----------------------+
| ip    | int(10) unsigned     |
| port  | smallint(5) unsigned |
| hash  | bit(64)              |
| date  | date                 |
+-------+----------------------+
```

Inserting the new crawl data every day line per line using a perl script took more time than the new data arrived. To insert bulk data in a very efficient way mysql provides the function `load data infile` that inserts the 1.5 million rows of a full crawl into an already filled table in only some seconds.

```
mysql> load data infile 'datafile' ignore
into table zonecrawl (@ipa, port, @dummy, @kid, @date)
set ip= INET_ATON(@ipa),  hash = unhex(substring(@kid,1,16)) ;

Query OK, 1574618 rows affected (4.20 sec)
Records: 1574618  Deleted: 0  Skipped: 0  Warnings: 0

mysql> select INET_NTOA(ip) as ipa ,port, hex(hash+0)
as kid from zonecrawl limit 5;
+----------------+-------+------------------+
| ipa            | port  | kid              |
+----------------+-------+------------------+
| 134.155.92.51  | 46725 | A4D05A132AC4CA99 |
| 83.44.89.63    |  4672 | 500F18E26E11F2B1 |
| 201.58.95.207  | 39162 | 5034FA2574DA07AE |
| 82.229.254.211 | 40440 | 510CE908EFF059FC |
| 82.253.80.148  | 34996 | 51FF9BA1A4A12320 |
+----------------+-------+------------------+
5 rows in set (0.06 sec)
```

The total size of the zone crawl table is of 360 million entries, which corresponds to 19 GB of data plus 25 GB of index. We created an index on all fields since in the beginning we did not know which queries we planed to execute, we wanted to "play" with the data. The only figures we wanted to get out of the full crawl was the KAD ID aliasing (Section 9.1.3). For this query a index would not have helped, the data accounts for 11 GB in 620 millions entries.

A query that asks for the point in time when a specific peer has been seen for the first and the last time runs in less than a second:

```
mysql> select min(time),max(time) from zonecrawl
where hash = unhex("5be81e38a288d9ce");
```

```
+---------------------+---------------------+
| min(time)           | max(time)           |
+---------------------+---------------------+
| 2006-09-23 00:00:00 | 2007-03-21 00:55:00 |
+---------------------+---------------------+
1 row in set (0.59 sec)
```

To compute the session times of all peers, all the data of the database needs to be returned in result. It is more efficient to build one query that returns a huge result set compared to a big number of smaller queries. Most database drivers for mysql, e.g. the ones for java or perl, are configured in a way that first mysql puts all the results in a temporary result table on disk before it ships the result. If your result set is bigger than the temp disk this does not work at all, and if the temp disk is huge it consumes a lot of time. The drivers have to be configured in such a way that no temp table is used and that every single line of the results set is delivered immediately. In java this can be done with the following commands:

```
stmt = conn.createStatement(java.sql.ResultSet.TYPE_FORWARD_ONLY,
       java.sql.ResultSet.CONCUR_READ_ONLY);
stmt.setFetchSize(Integer.MIN_VALUE);
```

APPENDIX B

Synthèse en Français

B.1 Tables de Hachage Distribuées

Les DHT (Tables de Hachage Distribuées) sont des systèmes répartis fournissant un service de recherche aux utilisateurs. Sont stockés dans la table des pairs (clé,valeur) et chaque utilisateur peut récupérer la valeur associée à une clé donnée. Ces couples sont répertoriés par tous les noeuds du système, de telle façon que la déconnexion d'un utilisateur n'entraîne pas de pertes importantes pour l'ensemble du réseau. Ainsi la DHT permet de gérer un nombre important d'utilisateurs, ainsi que l'arrivée de nouveaux noeuds, le départ d'anciens ou encore les erreurs de certains utilisateurs. Les opérations de base réalisables sur les DHT sont les opérations lookup(clé) et store(clé,valeur), permettant respectivement de récupérer la valeur associée à la clé et de stocker une clé et sa valeur associée dans la table de hachage distribuée.

Cette technologie est apparue avec la nécessité pour les réseaux pair- à-pair de se décentraliser. En effet, une telle structure ne nécessite pas de serveur central: un utilisateur qui arrive dans le système n'a qu'à se connecter à n'importe quel noeud du réseau pour en faire partie à son tour.

Le principe général des DHT peut en fait être expliqué de manière simple: aucun noeud d'un réseau pair-à-pair basé sur une table de hachage distribuée n'a une connaissance globale du réseau, mais il peut se rapprocher de la donnée cherchée jusqu'à la trouver (si cela est possible, bien évidemment).

Il faut noter qu'il existe d'autres architectures décentralisées mais non-structurées dans lesquelles ne sont pas prévus des mécanismes d'indexation de couple **(clé,valeur)**. Une requête de fichier dans un tel réseau est donc envoyée en gossip à toutes les machines du réseau, d'où un nombre important de messages envoyés entre les utilisateurs, et certains problèmes de réponses aux requêtes, que ce soit

au niveau du temps de réponse, relativement long, mais également au niveau de la quantité de réponses obtenues à une requête. L'utilisation de DHT permet de limiter l'envoi de messages concernant les recherches de fichiers: puisqu'il n'est pas nécessaire d'envoyer un message à tous, mais simplement de transmettre la requête de pair en pair jusqu'à arriver à obtenir le pair possédant l'information cherchée, en se rapprochant de lui à chaque "saut" dans le réseau.

Certains algorithmes de recherche dans les DHT, utilisés dans certaines implémentations, peuvent atteindre une complexité au pire en $O(log(n))$ pour un réseau constitué de n noeuds, d'où une complexité assez faible dans les meilleures implémentations, ce qui assure théoriquement un temps de réponse à une requête plus rapide que dans des réseaux non structurés.

La couche réseau fonctionne sur le principe que pour chaque noeud k, un autre noeud connaît soit directement ce noeud, soit un noeud plus proche de k qu'il ne l'est. Pour contacter le noeud k, il suffit à chaque fois d'envoyer le message au noeud que l'on connaît qui est le plus proche de k, qui le transmettra aussi au noeud le plus proche de k, et ainsi de suite.

Concernant les adresses des noeuds et les clés, ce qui correspond au terme de "hachage", on utilise une fonction de hachage que l'on peut appliquer aux adresses IP des pairs et aux chaînes de caractères qui correspondent aux ressources disponibles sur le réseau. Le résultat de la fonction de hachage est codé sur n bits (la valeur de n variant selon les protocoles, les valeurs les plus fréquentes étant 128 ou 160 bits) ce qui donne au total 2^n identifiants différents.

Le hachage de l'adresse IP donne l'identifiant d'un utilisateur, tandis que celui du nom d'un fichier donne la clé. Il s'agit ensuite de stocker de manière distribuée les couples (clé, identifiant) sur les noeuds du réseau, pour qu'à chaque ressource du réseau soit associée l'adresse de l'utilisateur possédant la ressource. La redondance dans le stockage est également introduite afin que le départ d'un noeud du système n'engendre pas la perte des meta données qu'il stocke et ne pas rende impossible l'accès à ces données.

Chaque noeud conserve des informations sur d'autres noeuds afin d'accélérer les recherches. Les algorithmes de recherche implémentes permettent qu'à chaque transmission de requêtes à un autre utilisateur, la distance séparant le demandeur et la clé cherchée soit divisée par deux. Dans les meilleures implémentations, cela permet d'atteindre une complexité logarithmique, ce qui est très inférieure aux complexités des systèmes complètement décentralisés fonctionnant par inondation.

B.1.1 Kademlia

Arrivé après les autres protocoles de DHT, le protocole Kademlia propose un réseau dont il précise la structure, les communications entre les noeuds et l'échange d'informations. La communication entre les noeuds du système se fait en utilisant le protocole UDP. L'intérêt du protocole Kademlia par rapport aux autres DHT est qu'il propose de calculer une distance entre les noeuds à l'aide de l'opérateur ou exclusif (XOR).

Les informations sont stockées dans des valeurs qui sont associées à des clés. L'ensemble des clés gérées par un noeud est en rapport avec son adresse, de telle sorte que si on connait une clé, la distance approximative entre l'utilisateur et le noeud possédant la clé recherchée peut être calculée par l'algorithme. Une recherche va donc se propager de voisin en voisin jusqu'à obtenir la clé voulue ou jusqu'à notification que la clé n'existe pas. Cette manière de propager les recherches permet au système d'avoir des recherches dont la complexité varie peu avec l'augmentation du réseau: par exemple, si la taille du réseau double, il suffira de demander l'information à un noeud de plus. On doit également relever que l'algorithme Kademlia est le plus utilisé par des applications pair-à-pair utilisées à grande échelle. En effet ce protocole a d'abord été implémenté par eDonkey2000 sous la forme du réseau Overnet et par eMule et son réseau KAD. On fait noter que ces réseaux sont incompatibles les uns avec les autres. Kademlia a ensuite été implanté chez le client BitTorrent Azureus, puis chez le client officiel BitTorrent. Kademlia est donc le protocole de DHT le plus utilisé avec notamment plus de 85 millions de personnes ayant d'ores et déjà installé eMule.

En effet, ce protocole de DHT a retenu l'attention de nombreux informaticiens grâce à sa mesure de distance basée sur des "ou exclusifs" qui en fait sur le papier l'une des tables de hachage distribuées susceptibles d'avoir les meilleures performances. De plus, on peut noter que Kademlia est la seule DHT utilisée actuellement dans des systèmes pair-à-pair de grande envergure, le logiciel de téléchargement de fichier eMule avec le réseau KAD, mais aussi comme système de recherche chez certains clients BitTorrent, comme Azureus.

Si le client eMule travaillait avec le réseau eDonkey basé sur des serveurs, il s'est démarqué de celui-ci en créant sa propre DHT, le réseau KAD, plutôt que de chercher à implémenter le réseau Overnet du client officiel. Deux DHT existent donc et sont incompatibles : KAD et Overnet. Cette dernière, a été condamnée par la justice à cesser ses activités de pair-à-pair, mais comme elle était totalement décentralisée, il fut impossible de l'arêter, et elle fonctionne actuellement encore avec les quelques utilisateurs qui y sont encore connectés.

La DHT Kad du client eMule est, elle par contre, sous licence de logiciel libre, et par conséquent, il est possible de regarder comment celle-ci est implémentée. Il faut

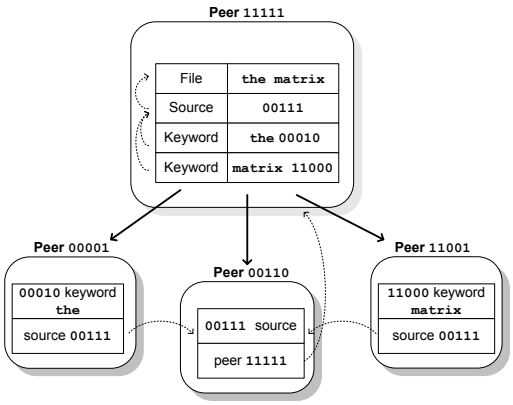


Figure B.1: Concept de la publication à deux couches en KAD.

toutefois remarquer qu'en cas de possibilité de réutilisation de la librairie Kad, il faudrait soit se connecter sur ce réseau et utiliser notre propre logiciel, soit modifier le code de KAD pour créer une nouvelle DHT propre à notre application mais qui fonctionnerait sur le même modèle que celle utilisée dans eMule.

Une **clé** dans un système pair-à-pair est un identifiant utilisé pour retrouver l'information. KAD se différencie entre deux types de clés:

- Une **clé source** qui identifie le contenu d'un fichier et qui est calculé en prenant le hash du *contenu* du fichier.
- Une **clé mot-clé** qui classifie le contenu d'un fichier et qui est obtenue en calculant le hash d'un seul mot du *nom* d'un fichier.

En KAD chaque clé n'est pas juste publiée sur un seul pair qui est le plus proche dans l'espace hash à la clé, mais sur 10 pairs différents. Les identifiants KAD de ces pairs doivent au moins avoir 8 bit en commun avec le mot clé. La zone ainsi définie est appelée *la zone de tolérance*.

Le processus de publication La figure B.1 montre un exemple du processus de publication à deux couches. Un pair veut publier un fichier avec le nom `the_matrix`. Il va être décomposé en deux mots clés, "the" et "matrix". D'abord les références aux fichiers originaux sont créées, ensuite les pointeurs des mots clés "the" et "matrix" sont publiés, qui pointent sur le fichier. Finalement la source elle même est publiée, et qui elle, pointe sur le pair qui l'a publié.

Les quatre types de messages les plus importants pour le routage, la publication et la recherche sont:

- `hello`: pour vérifier si l'autre pair est toujours en vie et pour l'informer de son existence, ainsi que de son adresse IP et l'identifiant KAD.

Algorithm 13: processus envoyer (exécuté une fois par crawl)

Data: *peer*: struct{IP addresse, numero de port, `kid`}
Data: *shared list* Peers = liste de *peer* elements
`/* la liste des pairs remplie par le processus recevoir et utilisée par le processus envoyer */`
Data: *int* position = 0
`/* la position dans la liste jusqu'à laquelle les pairs ont déjà été interrogés */`
Data: *list* ids = liste de 16 éléments *kid* choisis de manière appropriée

```
1 Peers.add(seed); /* initialise la liste avec les pairs graines */
2 while position < size(Peers) do
3     for i=1 to 16 do
4         destkid = Peers[position].kid ⊕ ids[i]; /* normalise le saut a la
              position du pair */
5         send route requests(destkid) to Peers[position];
6     position++;
```

Algorithm 14: processus recevoir (attend les messages `route response`)

Data: *message* mess = `route response` message
Data: *peer*: struct{IP address, port number, `kid`}
Data: *shared list* Peers = list of *peer* elements
`/* la liste partagée avec le processus envoyer */`

```
1 while true do
2     wait for (mess = route response) message; foreach peer ∈ mess do
3         if peer ∉ Peers then
4             Peers.add(peer);
```

- `route request/response(kid)`: pour trouver des pairs qui sont plus proches de l'adresse `kid`.
- `publish request/response`: pour publier une information.
- `search request/response(key)`: pour rechercher l'information avec la clé `key`.

B.1.2 Explorer les participants dans KAD

Une des contributions de cette thèse est le développement d'algorithmes qui permettent de connaitre tous les pairs dans KAD. Ces processus font usage de types de messages `hello` et `route request/response(kid)`. Le processus 13 envoie des requêtes de routages à tous les pairs découverts en demandant des adresses KAD

bien reparties dans l'espace de hachage. Le processus 14 reçoit les messages `route response`, en extrait les identifiants des pairs et les transmet à l'algorithme 13.

La vitesse de notre crawler nous a permis d'explorer tout l'espace des identifiants KAD, ce qui n'a jamais été fait avant. Un tel *crawl complet* nécessite environ 8 minutes. En 10 secondes, un million de pairs différent est découvert, après 50 secondes deux millions sont atteints, ensuite la vitesse s'affaiblit car une grande partie des pairs dans le système a déjà été découverte (Figure B.2). Un crawl complet de KAD produit à peu près 3 gigaoctets de trafic réseau entrant et sortant.

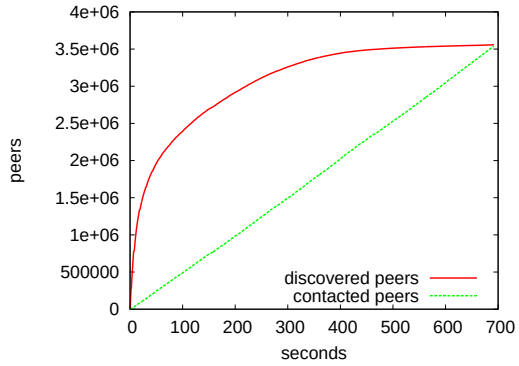


Figure B.2: Le nombre de pairs découverts s'approche de manière asymptotique du nombre total de pairs dans le système.

Pendant chaque *crawl complet* nous avons découvert entre 3 et 4.5 millions de pairs. Entre 1,5 et 2 millions de pairs ont pu être contactés directement et ont répondu.

Dans ce résumé, nous allons montrer seulement quelques uns des nombreux résultats que nous avons obtenus à l'aide du crawler, un outil unique.

Comme on peut le constater sur la figure B.3 le nombre de pairs varie selon un motif quotidien et hebdomadaire, atteignant le maximum pendant le weekend.

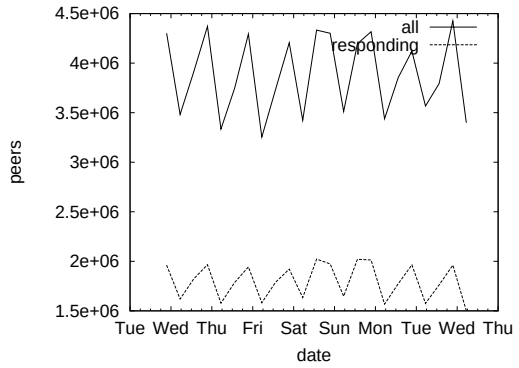


Figure B.3: Le nombre de pair KAD en ligne dans tout l'espace de hachage, selon l'heure du jour.

Dans la figure B.4 nous montrons la distribution des pays d'origine des pairs. L'Europe est le continent qui abrite le plus grand nombre de pairs (l'Espagne, la

France, l'Italie et l'Allemagne), pendant que la Chine est le pays avec le plus grand nombre de pairs. Moins de 15% des pairs viennent de l'Amerique (US, Canada et Amerique du Sud). Nous constatons également que la distribution du crawl complet est très proche de la distribution du crawl partiel d'une 256ième partie du réseau, ce qui confirme que les identifiants sont choisis de manière aléatoire.

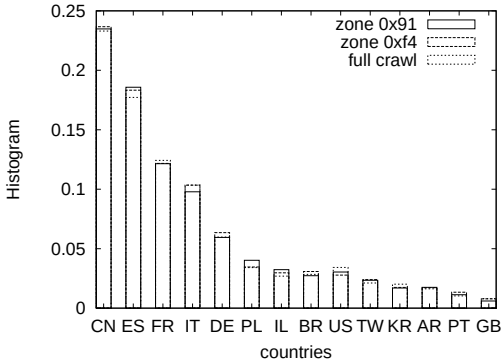


Figure B.4: Histogramme de la distribution géographique des pairs.

Sur la figure B.5, le nombre de pairs originaires de la Chine et de quelques pays Européens est dessiné. Il suit un motif quotidien, avec un pic autour de 21 heures, heure locale. Le décalage horaire de huit heures entre la Chine et l'Europe est nettement visible.

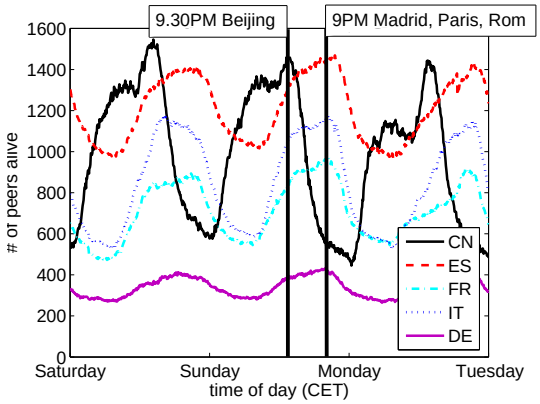


Figure B.5: Pairs en ligne selon leurs pays d'origine.

Dans la figure B.6 les temps de session sont représentés. Ceux des pairs découverts lors du *premier* crawl sont deux fois plus longs que ceux des pairs vus plus tard le même jour pour la première fois. Quand nous exécutons un crawl de KAD pour la toute première fois, nous avons une plus grande chance de tomber sur les pairs qui sont en ligne "la plupart du temps" plutôt que sur des pairs qui sont en ligne rarement. Ceci veut dire qu'un seul crawl ne peut fournir une image représentative des caractéristiques des pairs: nous sommes obligés d'exécuter des crawl régulièrement.

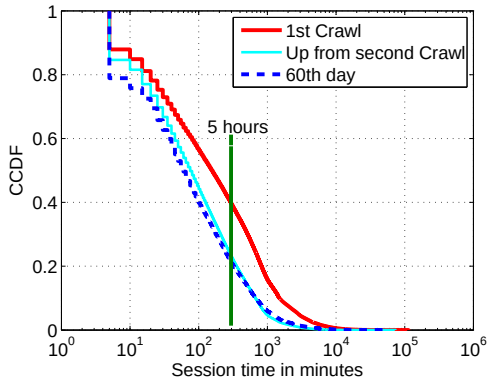


Figure B.6: La CCDF des longueur de sessions pour différents ensembles de pairs.

B.1.3 Explorer le contenu dans KAD

Afin d'analyser non seulement le comportement des pairs en KAD mais aussi le contenu, nous avons développé un espion, qui s'insère des milliers de fois dans la DHT et s'annonce à ses voisins. Ensuite il écoute les requêtes de recherche et de publication.

Nous avons lancé notre espion sur un 256ième de l'espace des identifiants KAD pendant 24 heures. Ceci est la durée minimum nécessaire pour détecter la publication de mots clés rares, car un mot clé n'est republié que toutes les 24 heures. Ainsi nous avons obtenu des résultats originaux:

- plus de 1,5 millions d'utilisateurs différents,
- plus de 1,4 millions de références différentes à des fichiers publiés,
- plus de 42.000 mots clés différents publiés, mais seulement 1.100 recherchés,
- chaque minute à peu près 1.000 requêtes de recherche, 10.000 requêtes de publication et 25.000 requêtes d'acheminent ont été reçues.
- La publication génère dix fois plus de messages que la recherche. De plus, les messages de publications sont dix fois plus grands que les messages de recherche. Ce qui fait une différence de deux ordres de grandeur pour le nombre d'octets transférés.
- La popularité des mots clés n'est pas distribuée uniformément, comme cela a été aussi observé dans d'autres DHT.

Si nous extrapolons ces chiffres à la taille de tout l'espace de hachage, nous obtenons le chiffre de 80 millions de fichiers partagés. Presque deux tiers de ces fichiers sont des fichiers audio, suivis par des fichiers vidéo, des logiciels, des document et finalement des images (Table B.0(a)). Le format de compression audio le plus utilisé est

le mp3 avec 87% des fichiers (Table B.0(b)). La moitié des fichiers sont encodés avec un taux de 128 kilo bits par seconde, un quart avec le taux plus élevé de 192kbps. La bataille des formats vidéos n'est toujours pas achevée: 34% de fichiers avi, 24% de fichiers wmv et 16% de fichiers mpg (Table B.0(d)). Pour les documents le format pdf l'emporte sans suprise avec 43% des documents (Table B.0(e)).

(a) Les fractions des différents types de fichiers.

file type	fraction
Audio	0.61
Video	0.15
Logiciel	0.11
Document	0.05
Image	0.05

(b) Les formats audio utilisés.

format audio	fraction
mp3	0.87
wma	0.02
mid	0.01

(c) Les taux de bits (kbps) utilisés pour la compression audio.

taux de bits (kbps)	fraction
128	0.50
192	0.26
160	0.06
320	0.05
256	0.02

(d) Les formats vidéo utilisés.

video format	fraction
avi	0.34
wmv	0.24
mpg	0.16
mkv	0.09
rmvb	0.04
asf	0.04
mpeg	0.03
rm	0.02

(e) Les formats de document utilisés.

document format	fraction
pdf	0.43
txt	0.17
doc	0.10
nfo	0.06
htm	0.02
html	0.01
ppt	0.01

Table B.1: Un aperçu des types et des formats de fichier retrouvés dans KAD. Certains types ou formats négligeables ont été omis, ce qui explique les sommes différentes de 1,00.

Les tailles de fichiers dépendent fortement des types de fichier. Pour l'audio à peu près 90% des fichiers ont une taille entre 2 et 10 megaoctets, alors que pour les fichiers vidéo la distribution est plus variable: la taille d'un film divx typique (aux environ de 730 megaoctets) et d'un film encodé avec la moitié de la résolution télé (360 MO) sont visibles dans la figure B.7. Plus de la moitié des documents ont une taille inférieure à 10 KO, il s'agit probablement de fichiers "lisez-moi".

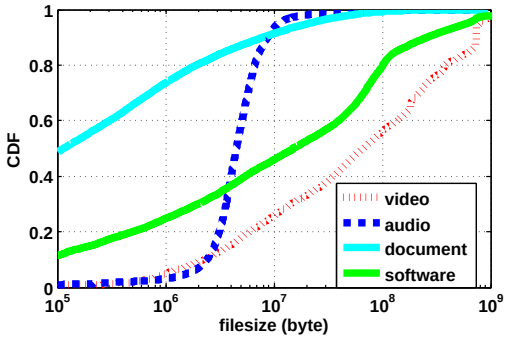


Figure B.7: La CDF des tailles de fichier selon le type de fichier.

B.2 Environnement Virtuel Partagé

Un environnement virtuel partagé est un espace généré par ordinateur(s) dans lequel plusieurs participants peuvent se rencontrer et interagir de telle manière que l'expérience vécue ressemble à celle qu'ils pourraient vivre dans le monde réel. On appelle entité n'importe quel objet indépendant dans le monde virtuel. Il s'agit d'un programme exécuté par une machine. Quand une entité est contrôlée par un humain, on dit que la représentation de l'entité dans le monde virtuel est l'avatar de l'humain qui la contrôle. On parle plutôt d'objet virtuel pour une entité contrôlée par l'ordinateur. Chaque entité possède une position dans le monde virtuel. C'est à cet endroit, virtuel, que sa représentation est observable par les autres entités. Traditionnellement, les mouvements sont autorisés: une entité peut bouger d'un point virtuel du monde à un autre. Par ailleurs, une entité peut apparaître dans un monde virtuel et en disparaître, aussi facilement que la machine qui l'héberge peut se mettre en marche ou s'arrêter. Un monde virtuel est donc dynamique: il se modifie au gré des événements qui l'affectent. Les environnements partagés impliquent plusieurs machines hébergeant une ou plusieurs entités susceptibles de modifier le monde virtuel à chaque instant. Ces machines disposent de la capacité de communiquer entre elles par des messages circulant sur un réseau, typiquement l'Internet.

La définition d'un système de communication pour une application telle qu'un monde virtuel massivement partagé comprend la gestion des liens de communication entre les entités et la gestion des messages échangés. Le système mis en place doit être conforme aux contraintes du réseau sur lequel repose l'application et respecter les spécifications de l'application.

Deux spécifications nous semblent fondamentales pour un monde virtuel massivement partagé : l'aptitude à passer à l'échelle et la cohérence.

- **Facteur d'échelle** : On considère qu'une application passe à l'échelle si son comportement n'est pas altéré lorsque le nombre de participants varie de plusieurs ordres de grandeurs (d'une dizaine à plusieurs millions

d'utilisateurs). On utilisera par la suite le néologisme scalable pour définir une application qui passe à l'échelle. Une application scalable se caractérise, entre autres, par une consommation des ressources physiques individuelles (le débit d'arrivée des messages, l'occupation de mémoire ou la charge de l'unité de calcul) indépendante du nombre total de participants.

- **Cohérence** : On dit qu'un environnement virtuel partagé est cohérent lorsqu'une scène virtuelle est perçue de manière identique par tous les participants qui l'observent. l'architecture d'un environnement virtuel partagé cohérent doit permettre à une entité désirant relater un événement de transmettre un message de telle manière que toutes les entités intéressées par cet événement le traitent simultanément.

Les architectures les plus courantes reposent sur des serveurs centraux. En relation avec toutes les entités, ils disposent d'une connaissance globale du monde. A la réception d'un message relatant un événement, ils doivent déterminer les entités qui sont intéressées par cet événement et leur transmettre le message. Ce mécanisme garantit une cohérence forte. Malheureusement, le nombre de messages que ces serveurs doivent traiter dépend du nombre de participants. De plus, ils peuvent être victime de défaillances susceptibles d'interrompre l'environnement virtuel.

Les architectures décentralisées sont destinées à corriger ces défauts. Chaque entité est directement connectée aux entités susceptibles de générer un événement dans la scène virtuelle observée. Comme la taille de cette zone est limitée, il est admis qu'une entité s'intéresse à un petit nombre d'entités, généralement indépendant du nombre total de participants. En conséquence, le nombre de messages simultanés que chaque entité doit traiter est à peu près constant, ce qui permet le passage à l'échelle. Malheureusement, la gestion de la mise en relation des entités reste un problème. Le plus fréquemment, une infrastructure de serveurs est chargée de détecter les entités les plus proches virtuellement et de permettre l'établissement de connexions entre elles. Même si le nombre de participants peut être élevé, le système ne passe plus à l'échelle. En outre, le déploiement et la maintenance de cette infrastructure génèrent un coût financier important tandis que le système est toujours soumis aux risques de défaillances.

B.2.1 Diagramme de Voronoi et triangulation de Delaunay

Le diagramme de Voronoi d'un ensemble S de n points de R^m est une partition de l'espace en n cellules représentant les zones d'influence des points de S : la cellule de Voronoi d'un point x de S est constituée de l'ensemble des points plus proches de x que de tout autre point de S. La triangulation de Delaunay de S est l'unique triangulation de S dont tout simplexe admet une boule circonscrite qui ne contient

aucun point de S (à part les sommets du simplexe). Ces deux objets, duaux l'un de l'autre, peuvent être linéarisés en augmentant la dimension de l'espace : ainsi, le diagramme de Voronoi et la triangulation de Delaunay de S sont des projections de polytopes de R^{m+1}. Les diagrammes de Voronoi et les triangulations de Delaunay dans R^m héritent donc des propriétés combinatoires des polytopes de R^{m+1}.

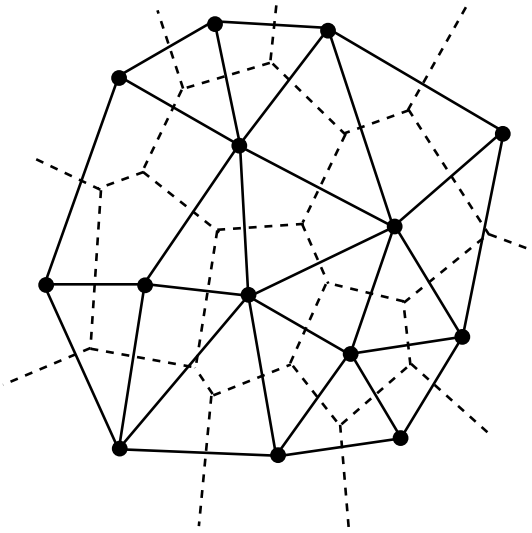


Figure B.8: Triangulation de Delaunay (lignes continue) et diagramme de Voronoi (ligne pointillée) en deux dimensions.

Le diagramme de Voronoi de S est un complexe cellulaire de R^m. Chaque cellule de dimension m correspond à un site x de S et est constitué de l'ensemble des points de E plus proches de x que de tout autre site de S. Chaque cellule de codimension 1 est constituée de l'ensemble des points équidistants de deux sites x et y de S et plus proches de ces deux sites que de tout autre site de S (cette cellule sépare les deux cellules correspondant à x et y respectivement), et ainsi de suite.

La triangulation de Delaunay de S est un complexe cellulaire qui forme une partition de l'enveloppe convexe de S. Lorsque les points sont en position générale, c'est-à-dire lorsqu'au plus $m+1$ d'entre eux appartiennent à une même sphère et au plus $k+1$ d'entre eux appartiennent à un même sous-espace affine de dimension k (pour tout $k < m$), la triangulation de Delaunay est un complexe simplicial : toutes les faces de Delaunay sont des simplexes.

Par ailleurs, le diagramme de Voronoi et la triangulation de Delaunay de S sont des complexes cellulaires duaux l'un de l'autre : pour toute partie R de S, la cellule de Voronoi $VorS(R)$ est non vide si et seulement s'il existe une boule circonscrite à R ne contenant aucun autre site de S (Figure B.8). L'application de $Vor(S)$ dans $Del(S)$ qui envoie $Vor_S(R)$ sur $Del_S(R)$ est une dualité entre complexes cellulaires (elle renverse l'ordre d'inclusion entre les faces).

B.2.2 Entretien d'un réseau pair-à-pair basé sur la Triangulation de Delaunay

Nous proposons des algorithmes légers, distribués, s'auto-organisant pour autoriser à un pair à s'ajouter au réseau. Le protocole est détaillé pour la triangulation de Delaunay en n dimensions.

Insertion de pairs

Nous considérons un nouveau pair z qui joint le système au temps t. Nous assumons que z a une position dans l'espace et connait au moins un pair dans le système.

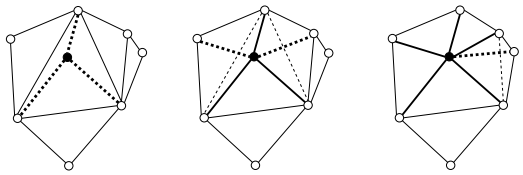


Figure B.9: Insertion d'un nouveau pair

Une technique connue en deux dimensions consiste à trouver le triangle englobant le nouveau pair et de séparer le triangle en trois nouveaux triangles. Finalement de manière récursive, tous les triangles adjacents sont testés pour déterminer si la procédure de retournement d'arêtes doit être appliquée.

Dans la figure B.10, l'arête (b, c) est remplacée par (a, z) parce que $\mathcal{C}(a, b, c)$ contient le nouveau pair z.

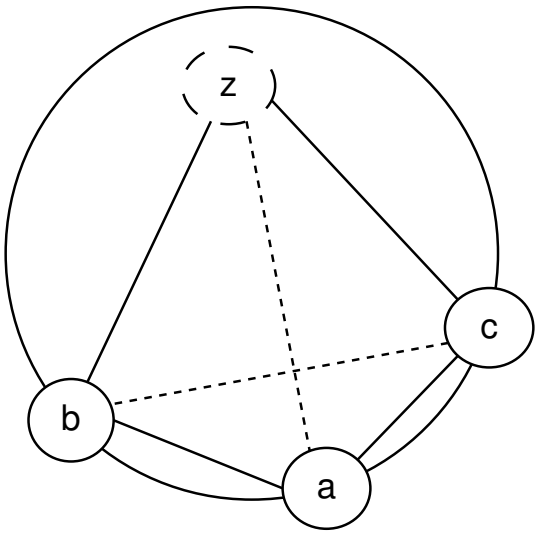


Figure B.10: Retournement d'arête

Peu d'articles ont étudié le comportement du retournement d'arêtes dans des espaces de n dimensions.

L'algorithme que nous proposons est séparé en trois tours. D'abord il faut trouver le d-simplex qui contient le nouveau pair. Ceci peut être fait avec une approche d'acheminement gourmand. Les deux tours suivants de l'insertion sont détaillés ci-dessous.

Diviser le simplex qui contient le nouveau pair Un pair inclus dans un d-simplex divise celui ci en $d+1$ d-simplex. Dans la figure B.11, un nouveau pair z appartenant au tétraèdre $T = (a, b, c, d)$ divise celui-ci en quatre tétraèdres $T_0 = (a, b, c, z)$, $T_1 = (a, b, d, z)$, $T_2 = (a, c, d, z)$ et $T_3 = (b, c, d, z)$.

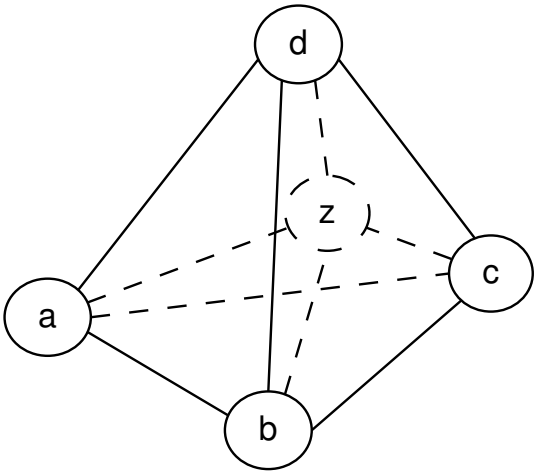


Figure B.11: Devision du tétraèdres contenant le nouveau pair

Le tour précédent se termine si le pair z reçoit une description des simplexes autour de sa position. Le prochain tour commence par la division de T en $d + 1$ d-simplex non superposés. Ensuite z enregistre ces simplex dans une liste $\mathcal{T}(z)$ où petit à petit tous les simplex impliqués sont ajoutés. Enfin z envoie un message `hello` à ses $d+1$ nouveau voisins. Ce message contient l'ancien d-simplex T et les d nouveaux simplex dans lesquelles la destination est impliquée. Dans l'exemple de la figure B.11, le pair z envoie à son voisin b un message `hello` qui contient T, T_0, T_1 et T_3.

Mécanisme de retournement d'arêtes récursives Nous considérons un pair a qui reçoit un message `hello` du nouveau pair z. Ce message contient un d-simplex T qui est obsolète et d nouveaux simplexes $T_1, T_2 \dots T_d$ qui contiennent les deux a et z.

Dans un cas normal, la triangulation est mise à jour en exécutant de manière récursive un mécanisme de retournement d'arêtes qui divise deux d-simplex en d d-simplex. Cette opération est appelée un $2 - d$ retournement. Dans la figure B.12, des

deux tétraèdres (a, b, c, z) et (a, b, c, e) résultent trois tétraèdres (a, b, z, e), (a, c, z, e) et (b, c, z, e).

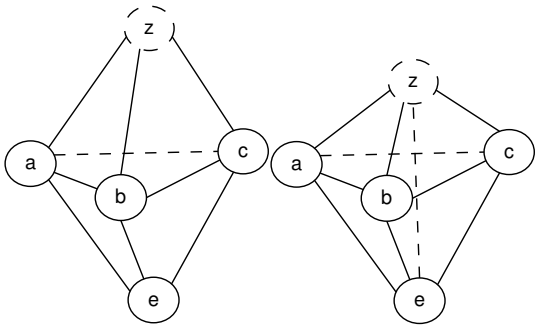


Figure B.12: Un $2 - d$ retournement

Le pair a doit d'abord déterminer le d-simplex $T_1' \in \mathcal{T}(a)$ de façon que T_1' et T_1 partagent un $(d-1)$-simplex en commun, par exemple $T_1 = (a, e_0, \ldots, e_{d-2}, z)$ et $T_1' = (a, e_0, \ldots, e_{d-2}, e)$. Le pair e peut être considéré comme l'opposé de z à travers de ce $(d-1)$-simplex. En deux dimensions e est l'opposé de z à travers l'arête (a, e_0) et en trois dimensions e est l'opposé de z à travers la face (a, e_0, e_1).

Si z appartient à l'hyper-boule autour de T_1', le pair a doit informer z que 1) le d-simplex T_1 doit être supprimé et que 2) e est un nouveau voisin. Ceci est atteint avec le message `detect` qui contient à la fois une description de e et T_1. Ensuite le pair a exécute le $2 - d$ retournement, qui crée d d-simplexes nommés T_{11}, T_{12}, ..., T_{1d}. Un de ces d-simplex ne contient pas a mais tous les autres oui. Entre temps T_1 et T_1' sont supprimés.

L'opération doit être réitérée avec les nouveaux simplex. Considérons T_{11}, le pair a doit trouver le d-simplex $T_{11}' \in \mathcal{T}(a)$ qui se partagent un $(d-1)$-simplex avec T_{11}. Ensuite a doit vérifier que le nouveau pair appartient à $\mathcal{C}(T_{11}')$. Si c'est le cas, a divise T_{11} et T_{11}' en d d-simplexes et envoie un autre message `detect` à z.

Le processus se termine lorsque a ne divise plus de d-simplex. Le pair a doit se disconnecter des pairs avec lesquels il ne partage plus de d-simplex in $\mathcal{T}(a)$. Dans cet algorithme tout les pairs impliqués doivent exécuter le mécanisme de retournement d'arêtes et envoyer un message `detect` à z.

L'algorithme 15 montre le pseudo code pour le traitement d'un message `hello`. Le premier test retourne le pair qui invalide T (lignes 2-3). Dans les autres cas, le pair reçoit la notification de d nouveaux d-simplexes $T_1 \ldots T_d$. Il les met dans une file d'attente fifo Q (lignes 5-6). Ensuite, il reçoit le d-simplex T_a (ligne 8). La fonction `share` prend en argument T_a et retourne un d-simplex qui partage un $(d-1)$-simplex avec T_a et le pair qui est opposé à z à travers ce $(d-1)$-simplex (ligne 9).

Si pour z le *hypersphère-test* est négatif le $(d-1)$-simplex T_a est sauvegardé (ligne 17). Le *hypersphère-test* vérifie si un pair est dans une hypersphère. En trois dimension un

Algorithm 15: `hello z` T T_1 ... T_d

```
1  if T ∉ T(a) then
2      f ← detectInside (T)
3      send "cancel T, f" to z
4  else
5      for i = 1...d do
6          Q.put (T_i)
7      while Q ≠ ∅ do
8          T_a ← Q.pop ()
9          T_b, e = share (T_a)
10         if z ∈ C(T_b) then
11             T_a1 ... T_ad ← split (T_a, T_b)
12             for i = 1...d do
13                 Q.put (T_ai)
14             remove T_b from T(a)
15             send "detect e T_a" to z
16         else
17             insert T_a in T(a)
```

pair p est dans $\mathcal{C}(a, b, c, d)$ si:

$$\begin{vmatrix} a_x & a_y & a_z & (a_x^2 + a_y^2 + a_z^2) & 1 \\ b_x & b_y & b_z & (b_x^2 + b_y^2 + b_z^2) & 1 \\ c_x & c_y & c_z & (c_x^2 + c_y^2 + c_z^2) & 1 \\ d_x & d_y & d_z & (d_x^2 + d_y^2 + d_z^2) & 1 \\ p_x & p_y & p_z & (p_x^2 + p_y^2 + p_z^2) & 1 \end{vmatrix} > 0$$

Dans le cas opposé, T_a est divisé et le processus récursif est mis en place en ajoutant les d-simplexes résultants à la file d'attente Q (ligne 12-13). Dans ce cas, un message `detect` est envoyé au nouveau pair z (ligne 15).

Dans ce résumé, nous ne détaillons pas la suppression de pairs.

Raccourci dans le réseau sous-jacent

Lorsque les réseaux structurés sur-jacents on été introduits, peu d'attention a été portée à la possibilité d'exploiter des information de proximité dans le réseau sous-jacent, le réseau physique, pour l'acheminement dans le réseau sur-jacent. En conséquence, un message envoyé à une destination utilisant l'acheminement dans le réseau sur-jacent va connaître une pénalité de latence. Afin de réduire cette pénalité, différentes propositions ont été faites.

Nous distinguons deux approches afin d'améliorer l'acheminement dans le réseau sur-jacent: i) faire mieux correspondre les deux réseaux et ii) construire un réseau logique qui a les propriétés de *petit-monde*. Dans le passage suivant, nous allons enrichir le réseau sur-jacent avec des liens additionnels de manière à ce qu'il ressemble à un petit-monde. En plus, nous allons donner priorité aux voisins physiques dans le réseau sous-jacent pour être élus comme **raccourci**.

Dans ce résumé, nous ne rentrons pas dans les détails ni du modèle ni de l'implémentation. Nous nous contentons de décrire la configuration de simulation et nous montrons quelques résultats.

Nous avons utilisé GT-ITM pour générer une topologie de réseau physique à deux niveaux, qui consiste en domaines et en noeuds à l'intérieur de ces domaines (Figure B.13). Parmi les noeuds qui n'ont qu'un seul lien, nous choisissons de manière aléatoire des noeuds qui vont participer au réseau sur-jacent.

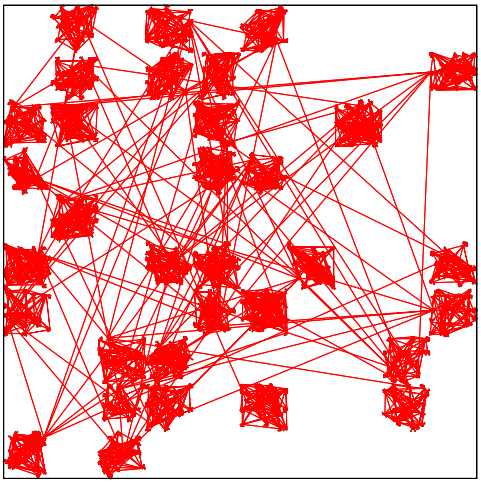


Figure B.13: Topologie de réseau (avec 8000 noeuds) organisée en deux niveau, générée avec GT-ITM.

La seconde étape de la simulation consiste au calcul de tous les chemins les plus courts dans cette topologie. La latence d'un plus court chemin est calculée en additionnant les distances d^u entre les noeuds sur le chemin.

Soit $p_i^u, p_{i+1}^u, \cdots, p_{i+h-1}^u, p_{i+h}^u = p_j^u$ le plus court chemin de pair p_i à pair p_j dans le réseau physique. La longueur de ce chemin est la somme des longueurs des dis-

tances Euclidienne des h sauts séparant les deux pairs:

$$\sum_{k=0}^{h-1} d^u(p_{i+k}, p_{i+k+1}).$$

La distribution des noeuds dans un monde virtuel n'est pas uniforme parce que les noeuds sont organisés dans des groupes. Afin d'obtenir ces groupes de noeuds, nous avons utilisé le vol de Lévy qui produit un chemin aléatoire à travers l'espace, en utilisant la distribution de Lévy pour déterminer la taille d'un pas. En regardant seulement les tournants de ce chemin, on obtient une distribution de noeuds organisés dans des groupes. (Figure B.14).

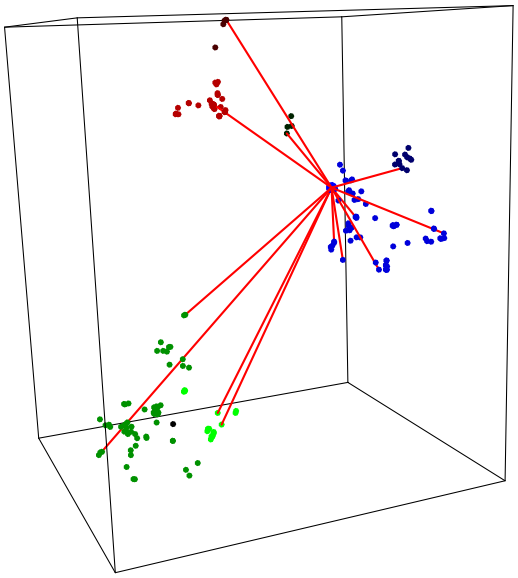
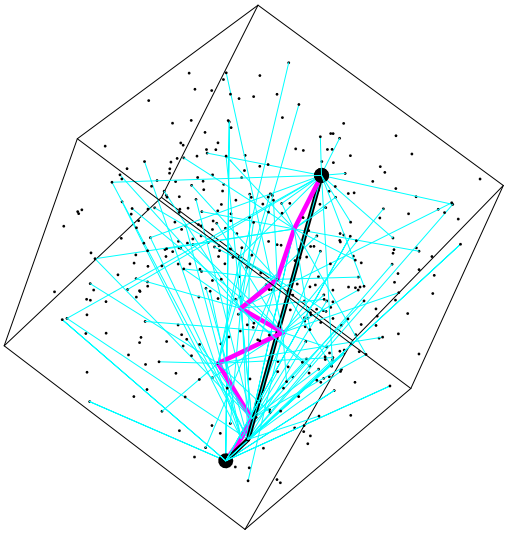


Figure B.14: Distribution de noeuds obtenue par un vol de Lévy.

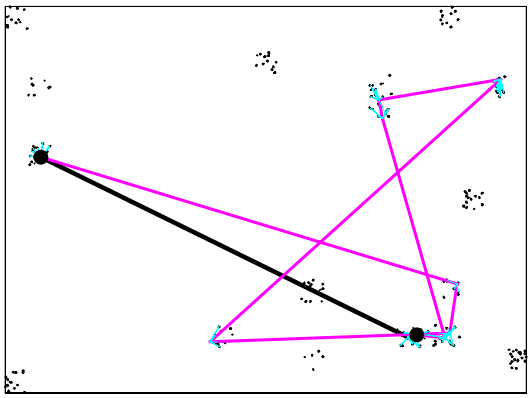
Nous avons effectué des simulations avec des distributions de noeuds uniformes et regroupés. L'exemple dans la figure B.15 contient 1296 noeuds et 3680 arêtes dans le réseau physique et 397 noeuds participent au réseau logique superposé. Le chemin affiché est le plus long, celui qui profite le plus des raccourcis.

Si nous comparons les deux chemins pris au niveau physique, nous constatons que les raccourcis ajoutés peuvent être très efficaces: le nombre de sauts a été réduit de 6 à 3 et la latence de 11049 ms à 3418 ms. Le chemin le plus court dans le réseau physique a une latence de 2471 ms. Seulement le premier des sauts faits est un raccourci, il s'agit du plus long dans la figure B.15(a).

Dans cette synthèse, nous n'abordons pas le regroupement des participants.



(a) Chemin pris dans le réseau logique, depuis le point en haut au point en bas de la figure. Les lignes fines indiquent les raccourcis des noeuds non visités.



(b) Chemin pris dans le réseau physique, depuis le point sur la droite au point sur la gauche. En gris le chemin sans raccourcis, en noir le chemin en prenant avantages des raccourcis.

Figure B.15: Chemin pris dans le réseau logique et le réseau sous-jacent.

List of Publications

[1] Moritz Steiner and Ernst W. Biersack. A fully distributed peer to peer structure based on 3D Delaunay Triangulation. In *Proceedings of Algotel - Septièmes Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications*, pages 93–96, May 2005.

[2] Moritz Steiner and Ernst W. Biersack. DDC: A Dynamic and Distributed Clustering Algorithm for Networked Virtual Environments based on P2P networks. In *Proceedings of CoNEXT (Poster)*, Toulouse, France, October 2005.

[3] Moritz Steiner and Ernst W. Biersack. DDC: A Dynamic and Distributed Clustering Algorithm for Networked Virtual Environments based on P2P networks. In *Proceedings of the 9^{th} IEEE Global Internet Symposium in conjunction with IEEE Infocom*, Barcelona, Spain, April 2006.

[4] Moritz Steiner and Ernst Biersack. Shortcuts in a Virtual World. In *Proceedings of CoNext*, 2006.

[5] Moritz Steiner, Ernst W. Biersack, and Taoufik En-Najjary. Actively Monitoring Peers in Kad. In *Proceedings of the 6^{th} International Workshop on Peer-to-Peer Systems (IPTPS)*, 2007.

[6] Moritz Steiner, Wolfgang Effelsberg, Taoufik En-Najjary, and Ernst W. Biersack. Load Reduction in the KAD Peer-to-Peer System. In *Fifth International Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P 2007)*, 2007.

[7] Moritz Steiner, Taoufik En-Najjary, and Ernst W. Biersack. A Global View of KAD. In *Proceedings of the Internet Measurement Conference (IMC)*, 2007.

[8] Moritz Steiner, Ernst W. Biersack, and Taoufik En-Najjary. Exploiting KAD: Possible Uses and Misuses. *Computer Communication Review*, 37(5), October 2007.

[9] Moritz Steiner, Damiano Carra, and Ernst W. Biersack. Faster content access in kad. In *Proceedings of the 8^{th} IEEE Conference on Peer-to-Peer Computing (P2P)*, Aachen, Germany, September 2008.

[10] Thorsten Holz, Moritz Steiner, Frederic Dahl, Ernst W. Biersack, and Felix Freiling. Measurements and Mitigation of Peer-to-Peer-based Botnets: A Case Study on Storm Worm. In *First Usenix Workshop on Large-scale Exploits and Emergent Threats (LEET)*, San Francisco, CA, USA, April 2008.

[11] Moritz Steiner, Ernst W. Biersack, and Taoufik En-Najjary. Long Term Study of Peer Behavior in the KAD DHT. *Accepted for Publication in IEEE/ACM Transactions on Networking*, 2008.