Papnet: An order-preserving and latency-aware P2P Overlay and its Applications

vorgelegt von
Martin Raack, MSc
aus Cottbus

Von der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften
- Dr. rer. nat. -

genehmigte Dissertation

Tag der wissenschaftlichen Aussprache: 12. Januar 2012

Berlin 2012

D83

Danksagung (Acknowledgement)

Ich möchte mich zuvorderst bei meinem Betreuer Odej Kao bedanken, der mir über die letzten Jahre stets mit guten Ratschlägen unterstützend zur Seite stand, hervorragende Arbeitsbedingungen bot und hiermit letztlich diese Dissertation erst ermöglicht hat. Weiterer Dank gebührt Jörg Nolte, der sich zur Begutachtung bereit erklärte, wie auch dem Vorsitzenden des Prüfungsausschuss Volker Markl.

Des Weiteren danke ich den Kollegen des CIT mit denen ich über die vergangen Jahre arbeiten, diskutieren und auch viele erfreuliche ausserdienstliche Abende verbringen durfte, insbesondere Felix Heine, Dominic Battré, André Höing, Philipp Berndt, Björn Lohrmann, Matthias Hovestadt, Alexander Stanik, Mareike Strüwing, Andreas Kliem sowie meinem Teilzeit-Büro-Partner Daniel Warneke. Auch danke ich Christian Würtz und Stefan Menzel, die mich bei meiner Forschung unterstützten und deren Abschlussarbeiten ich betreuen durfte.

Ich danke meinen Eltern Siegfried und Irene Raack, die mich schon sehr früh mit der Informatik in Berührung gebracht, begeistert und gefördert sowie blockierte Telefonleitungen und die damit verbundenen Kosten ertragen haben. Weiterhin schulde ich meinen lieben Geschwistern Thomas, Andrea und Wolfgang grossen Dank.

Nicht zuletzt möchte ich auch den Menschen danken, die mich über die Jahre begleitet und geprägt haben, namentlich Martin Sabransky, Axel Vick, Robert und Anika Schuppenies, Jacky Weinert, Dr.Steffen Pepper, Stephanie Schneider, Kati Finzel, Maik Donath, Maria Gosdschan, Marlen Kaletka, David Geier, Christian Loos, Eric Werner, Frank Scheffel sowie Cindy Mentzel und Anne-Katrin Fischer.

Abstract

This thesis describes the development of a new P2P Overlay called *Papnet*, which combines the advantages of Distributed Hash Tables with those of Order-Preserving P2P Overlays. Papnet does not require any hashing and is thus able to store object keys in a sorted manner. This enables the efficient processing of range queries as well as the implementation of a load balancing technique that guarantees a constant global load imbalance ratio.

Papnet is latency-aware. Given a uniform distribution of latencies it is able to route between arbitrary nodes within only twice their direct latency, independent of the actual network size. We show, that in contrast to other Overlays Papnet is able to guarantee a fast convergence towards latency-optimal routing links.

As a direct application of Papnet we present a new algorithm to process window- and k-nearest-neighbor queries on spatial point data, which is able to scale asymptotically linear with the total query load. In contrast to existing solutions, the construction and maintenance of an explicit distributed spatial structure is not required.

Zusammenfassung

Diese Dissertation entwickelt ein neues P2P Overlay Netzwerk namens *Papnet*, welches die einzelnen Vorzüge bestehender Typen von P2P Overlays vereint und erweitert. Im Unterschied zu den meisten der existierenden P2P Overlays erfordert Papnet kein Hashing und kann somit einzelne Datenelemente sortiert nach einem Primärschlüssel speichern. Dies ermöglicht sowohl effiziente Bereichsanfragen als auch die Implementierung einer Lastbalancierungs-Technik welche ein konstantes Lastungleichgewicht garantiert.

Papnet berücksichtigt Nachrichtenlaufzeiten und ist bei globaler Gleichverteilung der Latenzen in der Lage die zuständigen Knoten zu beliebigen Objektschlüsseln unabhängig von der Größe des Netzwerks in circa zweifacher direkter Laufzeit zu erreichen. Wir zeigen, dass Papnet im Unterschied zu bestehenden P2P Lösungen hierbei eine schnelle Konvergenz zu einem Laufzeit-Optimum garantiert.

Als direkte Anwendung von Papnet stellen wir einen neuen Algorithmus zur verteilten Bearbeitung von geospatialen Daten vor. Dieser ermöglicht eine praktisch lineare Skalierung mit zunehmender Anfragelast und erfordert im Gegensatz zu existierenden Lösungen nicht den Aufbau und die Pflege einer speziellen verteilten räumlichen Datenstruktur.

Contents

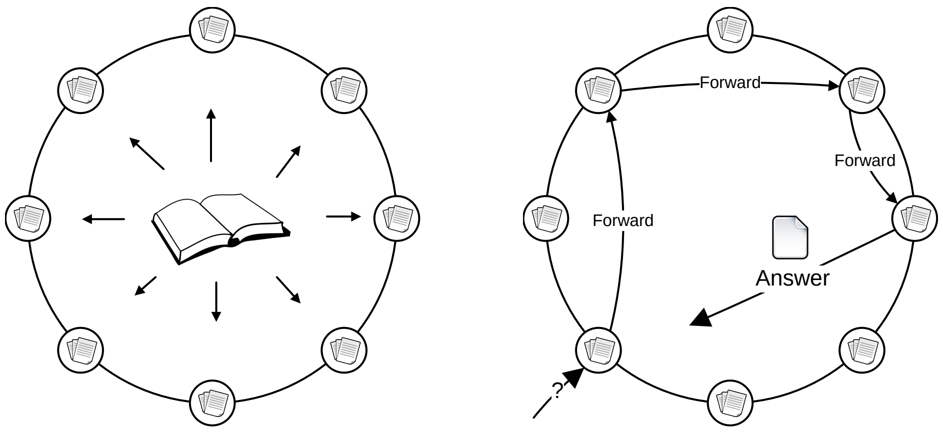
1 Introduction

Contents

Having a special focus on systems that incorporate large quantities of machines, many structured Peer-to-Peer (P2P) overlay networks have been developed over the past years. Typically, P2P networks consist of large numbers of autonomous computers that share resources to reach a common goal. They feature redundancy and automated repair mechanisms, enabling them to tolerate machine failures because there is no single-point-of-failure in the system. Each peer can join or leave at any time without serious impact on the other peers or the common goal. P2P technologies can be used in various scenarios like distributed data management, resource discovery, semantic reasoning, audio and video streaming, keyword search, spatial processing and even social networking. P2P Overlay networks provide data-oriented address spaces which are independent of the actual underlying physical network. They enable the efficient lookup of arbitrary addresses using only a small number of forwarding steps (so-called hops), while each participating node is required to know only a small portion of the global network state.

An important P2P application is distributed data management, where each peer stores only a small fraction of the overall data but still has efficient access to the complete set of data. Such distributed P2P databases are often realized through Distributed Hash Tables (DHT). A DHT stores key-value pairs in a very large hash table that is hold distributed over all peers participating in the network and each peer is responsible for a well defined part of the hash table's id space. DHTs allow for routing towards arbitrary nodes within a bound number of hops. When nodes join or leave the network, the routing information is updated through maintenance protocols.

An example DHT application is a distributed phone book stored in a system comprising n nodes. Each phone book record consists of a key representing a person's name and a value representing his/her phone number. The DHT partitions the phone book into disjunctive "chapters" and each node in the network is responsible for exactly one such chapter. A user can look up a phone number by issuing a query for a person's name at an arbitrary node of the network.



(a) Partitioning into "Chapters"   (b) Phone number search

Figure 1.1: DHT Phone Book Example.

The DHT structure and routing algorithm ensure that the node being responsible for the "chapter" of the corresponding record (if any) will be found within an efficiently small number of message forwarding steps. Figure 1.1 illustrates the phone book DHT.

DHTs usually provide implicit storage load balancing by using hash functions to map object names to identifiers within the id space, resulting in a uniform distribution. Due to the need for hash-mapping, it is only possible to retrieve elements if their exact key is known. DHTs thus lack an important feature: they do not natively provide so-called *range-queries*.

In the phone book example, this means that the "chapters" are not chapters in the classical sense, i.e. ordered names ranging from some letter A to some letter B. Instead, each "chapter" may contain arbitrary names. Queries can be answered quickly if the full name of the person is known, but given only the first name, it is impossible to lookup all matching entries, since their hashed identifiers cannot be predicted solely from the first name. Even though two entries "Smith Joe" and "Smith John" are very similar, they will get mapped to completely distinct keys in the finite key space of the DHT, making it impossible to answer wildcard- and range-queries such as *get("Smith Jo*")* and *getRange("Smith", "Snyder")*.

Since each node in the network should be responsible for an equal share of records, care must be taken when partitioning the phone book into "chapters". In DHTs, a reasonable load balance is accomplished implicitly, by utilizing the aforementioned hash function that distributes keys uniformly in key space.

Thus, equally sized portions of the key space can be expected to contain equal amounts of data records, but the global imbalance (ratio of most- to least-loaded node) can still vary in magnitudes of $O(\log n)$ [1]. Even worse, there is a problem with this strategy when it comes to our phone book example, where certain names are far more popular than others, e.g. "John Smith". To be retrievable, all "John Smith" entries need to be stored at the very same (hash-)key, which inevitably creates so-called *load hotspots* that cannot be split.

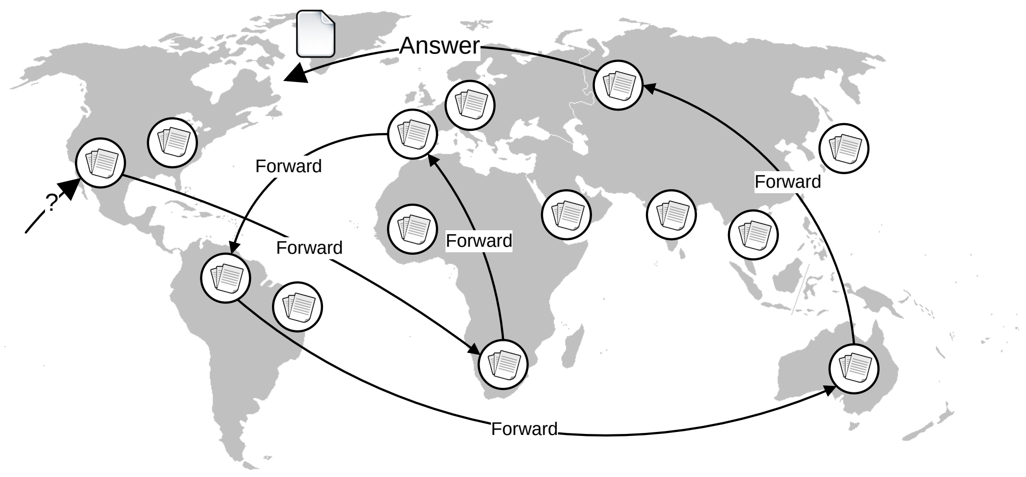


Figure 1.2: Non-Latency aware DHT routing example.

Another important property of a P2P Overlay is its latency awareness. Without routing tables being optimized for proximity, the total routing path latency is likely to grow proportionally to the path length (number of hops).

Imagine our phone book system consists of n peers that are spread across the world. Then even if the utilized P2P system guarantees an efficient routing path of length $O(\log n)$, each such routing step may result in a message transmission from one continent to another, resulting in a very bad total path latency. Figure 1.2 illustrates the routing in such a non-latency-aware DHT.

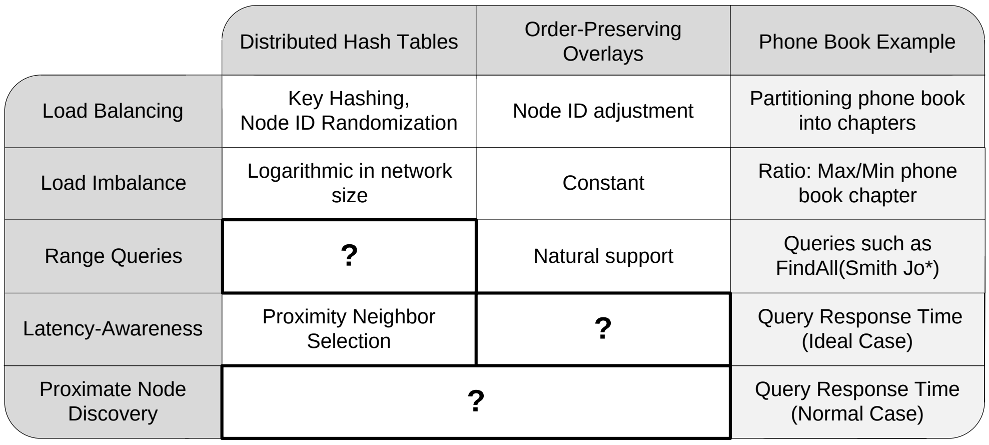
Fortunately, there exist techniques such as *Proximity Neighbor Selection* which enable DHTs to prefer low-latency routing links. Assuming a uniform distribution of latencies in the network, using this optimization technique the total path latency can be reduced to a *constant* factor of the direct latency between source and target node, independent of the actual network size.

1.1 Problem Definition and Contribution

Distributed Hash-Tables, which enable the distributed storage and retrieval of (key, value)-pairs, are the most common application of structured P2P Overlay networks. Chord [2], Pastry [3], and Tapestry [4] are popular overlay networks that allow for the implementation of a DHT. However, applications such as the phone book example do require range queries, which are not supported by the above networks. Further, these Overlays are not capable to relax load hotspots that arise from the lack of range query support due to key hashing.

There is a second type of Overlay networks, allowing for the implementation of DHT-like distributed data stores, which do not require hashing and thus can store keys in an order-preserving way. Example networks are SkipNet [5] and Chord# [6], which both support range queries. Further, they allow for an integration of sophisticated load balancing such as the On-Line load balancing proposed in [7] by Ganesan et al., which provides a *constant* global imbalance ratio. However, existing order-preserving networks establish their Overlay topology strongly deterministic and do not allow for *Proximity Neighbor Selection* as in classic DHTs. Thus, the total path latency depends on the total network size.

Figure 1.3 depicts the properties of both Overlay types and links them to the phone book example requirements. The figure also lists the property *Proximate Node Discovery.* Note that even though Proximity Neighbor Selection enables the optimization of routing links for proximity in DHTs, there is no guarantee that any node will eventually learn about the actual existing most proximate peers.



	Distributed Hash Tables	Order-Preserving Overlays	Phone Book Example
Load Balancing	Key Hashing, Node ID Randomization	Node ID adjustment	Partitioning phone book into chapters
Load Imbalance	Logarithmic in network size	Constant	Ratio: Max/Min phone book chapter
Range Queries	?	Natural support	Queries such as FindAll(Smith Jo*)
Latency-Awareness	Proximity Neighbor Selection	?	Query Response Time (Ideal Case)
Proximate Node Discovery	?		Query Response Time (Normal Case)

Figure 1.3: Properties of Overlay Types in relation to Phone Book Example

The main part of this thesis, Chapters 3-5, covers the problem of how to combine the advantages of classic DHTs with those of order-preserving Overlays. In particular, we will tackle all of the three question marks depicted in Figure 1.3.

Chapter 3 presents a solution to the problem of missing range query support in classic DHTs. We extend the classic DHT Pastry [3] to avoid key hashing, so that the resulting overlay is capable of storing data in a totally ordered manner. Therewith, range queries become possible, as well as constant-imbalance load balancing. We demonstrate the feasibility of our DHT extension in an evaluation with simulations and also discuss encountered problems.

In Chapter 4 we pursue the complementary direction, introducing Proximity Neighbor Selection into order-preserving Overlays. We present a detailed description of a new Overlay network called *Papnet*, which is a hash-free and latency-aware P2P Overlay that supports range-queries and realizes an infinite alphanumeric address space that can be used to store arbitrarily skewed data. Further, it is suitable for constant-imbalance load balancing. We evaluate Papnet in a real distributed environment with a network comprising 50,000 nodes.

In Chapter 5, we will focus on the active discovery of proximate neighbors. We present a new low-cost discovery protocol utilizing the Papnet topology and show, that hash-free networks such as Papnet can achieve very low path latencies, comparable to those of the well-known latency-aware but hash-based overlay Pastry. Moreover, we show that our approach guarantees eventual convergence to optimal routing tables, a property not yet achieved in previous Overlays.

Chapter 6 presents an application that benefits from Papnets latency awareness, order-preserving properties and load distribution capabilities. We propose a new distributed system based on a P2P architecture to store and process spatial data, in particular so-called *window-* and *k-nearest-neighbor* queries. Our system is very simple in that it solely manages a range-partitioned linear data space defined by a Hilbert Curve mapping and neither requires explicit hashing, clustering or the maintenance of a dedicated distributed spatial structure at all. Our main focus is on the inherent quad-tree structure of the 2d Hilbert Curve and how it suffices to efficiently evaluate nearest-neighbor queries in a distributed manner. We evaluated our approach using real-world data from Open Street Map and demonstrate our system to scale asymptotically linearly with the network size.

Parts of this thesis have been published in the following publications:

1. **Dominic Battré, André Höing, Martin Raack, Ulf Rerrer-Brusch, Odej Kao**
   Extending Pastry by an Alphanumerical Overlay
   In: Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'09), pp.36–43, acceptance rate 21%, 2009
2. **Martin Raack, Dominic Battré, André Höing, Odej Kao**
   Papnet: A Proximity-aware Alphanumeric Overlay Supporting Ganesan On-Line Load Balancing
   In: Proceedings of the 15th International Conference on Parallel and Distributed Systems (ICPADS'09), pp.440–447, acceptance rate 29.5%, 2009
3. **Martin Raack, Christian Würtz, Philipp Berndt, Odej Kao**
   A hash-free overlay with eventually optimal locality
   In: Proceedings of the Annual International Conference on Network Technologies and Communications (NTC'10), pp.N59-N64, 2010
4. **Martin Raack, Odej Kao**
   Scalable distributed processing of spatial point data
   In: Proceedings of the 17th International Conference on Parallel and Distributed Systems (ICPADS'11), acceptance rate n/a yet, to appear, 2011

2 Peer-to-Peer Basics

Contents

The term *Peer-To-Peer* (P2P) refers to distributed systems consisting of autonomous participants that implement both client and server functionality. While there is no single comprehensive definition that characterizes P2P systems in all aspects, the term often describes the paradigm of decentralized network organization comprising functionally equal entities, which is a complementally approach to classic Client/Server architectures. This chapter will provide an introduction to P2P networks, starting with a brief P2P history.

2.1 History

P2P networks of the so-called first generation were systems based on classic Client/ Server architectures that only partly utilize P2P mechanisms. The most prominent representative was Napster[1], which enabled users to share music files. The actual file transfers took place using direct TCP connections between the end users, so large numbers of parallel transfers without server involvement became possible. A major disadvantage of Napster was the necessity of a central server instance that managed the user accounting and the central file index. Due to this centralization, Napster could almost instantly be shut down after judicial decision. Without the central server and its provided search functionality, Napster ceased existence.

After recognizing the vulnerability of centralized systems, research towards decentralized failure-resilient networks flourished. The first representatives of such

[1]http://www.napster.com

systems were unstructured networks with arbitrary ad-hoc topologies like Gnutella [8] and FastTrack [9]. A major disadvantage of unstructured P2P networks is the absence of performance and availability guarantees. They neither allow to specifically address a certain participant in the network, nor do they enable lookup paths of bounded length. Search operations in these networks always involve a flooding of the global network, which results in large communication overhead. Thus, these networks are unable to scale with query load.

In contrast to unstructured Networks, so-called *structured* P2P networks provide a well-defined topology and common id space. This enables an efficient routing towards arbitrary IDs as well as to provide performance guarantees. Most such networks realize a so-called *Distributed Hash Table* (DHT) which allows to store and efficiently retrieve data objects in a globally partitioned address space. DHTs utilize so-called *hash functions* [10] such as SHA [11], which map data object names onto keys in the global address space. The usage of hash functions is necessary to ensure a uniform distribution of both data objects and participants (nodes) in key space. Section 2.3 will provide a classification of structured second generation P2P overlays.

2.2 Applications

P2P Overlays are nowadays used in many different application scenarios. The probably most well-known application is the *Voice-over-IP* (VoIP) software Skype [12], which is presumed to be based on an unstructured P2P Overlay similar to the *FastTrack* network, which has previously been developed by the Skype founders. Since Skype uses a proprietary protocol, little is known about the exact protocol, but it seems to incorporate so-called *Super-Peers* with good latency and large bandwidth which serve as Hubs for the normal users. While Skype is the most prominent P2P VoIP example, there are further approaches such as OpenVoIP [13] which uses a DHT directory service, or extension to SIP (Session Initiation Protocol [14]) servers using a DHT like Chord as proposed in [15].

Another field of application is distributed data storage. Dynamo [16] is a distributed key/value store based on a P2P Overlay that provides $O(1)$ routing complexity, but on the other hand each node has linear state complexity of $O(n)$, i.e. each node knows *all* other nodes. This approach is feasible for small clusters of nodes but not at large scale. Dynamo has been developed and published by Amazon, but no publically available implementation has yet been released.

Cassandra [17] is another key/value store with $O(1)$ routing similar to Dynamo, that has been developed at Facebook and was publicly released as open source.

It since has become a major project of the Apache Foundation[2] and is/has been in productive use at Facebook, Twitter, Digg, Reddit, Rackspace, Cisco and a lot more companies [18].

Large scale systems however require a sub-linear node state to coop with increasing system size. Current clusters still comprise only low numbers of nodes, so the need for routing path guarantees *and* efficient node state is still quite low. However, as hardware costs will decrease and demand for compute time as well as storage will increase, future systems will eventually need to scale to very large numbers. A non-commercial structured Overlay that is already used by up to several hundreds of thousands of people each day is Kademlia.

Kademlia [19] is a prefix-oriented structured overlay that has been used in several P2P file sharing applications, e.g. the *eMule Kad* network, *Overnet* and in *Bittorrent* as a distributed *Tracker* [20]. Further, even remotely controlled *Bot Networks* have been observed to employ Kademlia [21]. Its failure resistance due to its redundant link maintenance and routing, as well as its latency optimizations and scalability make it an ideal candidate in scenarios that consist of large numbers of geospatially distant nodes.

In general, P2P networks are ideally suited to provide a foundation for content distribution systems. A survey by Liu et al. [22] lists several video streaming solutions based on P2P Overlays. Some construct multicast tree structures e.g. SplitStream [23] which the authors apply to Pastry; others facilitate mesh structures for efficient distribution without depending on a particular P2P substrate, e.g. Bullet [24].

There have also been efforts to realize fault-tolerant and censorship-resistant, decentralized replacements for the currents internets *Domain Naming System* [25]. One such system is DDNS [26], which is based on DHash[27] – an extended Chord network that provides latency-awareness. Another example is CoDoNS [28] that utilizes Beehive [29], a framework for classic DHTs that achieves $O(1)$ routing complexity through data replication. A comparison that includes further systems is given in [30].

Another promising application scenario is a distributed social network. While there exist plenty of popular centralized solutions like MySpace[3] or Facebook[4], there are also numerous approaches to decentralize such services to respond to Orwellian fears of surveillance and to retain control over one's own data. One example is LifeSocial [31] which is based on the Pastry Overlay. Another network is PeerSoN

[2]http://cassandra.apache.org
[3]http://www.myspace.com
[4]http://www.facebook.com

[32] which uses OpenDHT/Bamboo [33], a structured Overlay also based on the Pastry geometry. Safebook [34] is yet another example using the Kademlia Overlay [19].

While the applications mentioned above are based on one-dimensional Overlays, there are also lots of applications utilizing multi-dimensional Overlays. An example is PRoBe [35], which enables multidimensional keyword searches and range queries by implementing a multidimensional constant node degree topology similar to CAN [36].

Another multidimensional application are geographic data stores, where each spatial axis corresponds to a dimension in the overlay. SONAR [37] implements such a functionality and is based on a multidimensional logarithmic degree Chord# Overlay [6]. But multidimensional application scenarios not necessarily require multidimensional Overlay topologies. Instead, P2P information retrieval system such as the one presented in [38] utilize so-called space-filling-curves to map domains of higher dimension down to a single one. We will elaborate more on multidimensional applications and distribution approaches in chapter 6, which presents a spatial data processing service based on the new Overlay Papnet.

2.3 Classification

There are several types of structured P2P overlays and it is not trivial to classify the various structures, because they exhibit different state and routing complexities, have different requirements and provide different features. An essential distinction can be made regarding space complexity, where we distinguish three classes of overlays: networks with *linear*, *constant* and *logarithmic* node degree. Table 2.1 gives an overview over a representative selection of network structures from these classes and their specific properties. Further comparison studies that classify more networks and overlay types can be found in [39] and [40].

Linear Degree Networks

In networks with linear degree, each node knows *all* other nodes. The state of the entire system is periodically exchanged between the nodes by means of so-called *Gossip protocols* [41], [42]. These protocols ensure that state updates disseminate throughout the network in an epidemic way, reaching each node in a time that is logarithmic in the number of nodes in the system [42]. Since each node knows the entire network, the routing complexity is in $O(1)$ which guarantees the shortest possible latency. Also, these systems exhibit very good fault tolerance properties.

Overlay Network	**Space Compl.**	**Routing Compl.**	**Dimensions, Key Space**	**Assump- tions**	**Fault Tolerance**	**Latency Aware**
Cassandra	$O(n)$	$O(1)$	1, finite	Hashing	Max	Max
Dynamo	$O(n)$	$O(1)$	1, finite	Hashing	Max	Max
CAN	$O(1)$	$O(\sqrt[d]{n})$	d, finite	Hashing	Low	Low
Viceroy	$O(1)$	$O(\log n)$	1, finite	Hashing	Low	No
Koorde	$O(1)$	$O(\log n)$	1, finite	Hashing	Low	No
Chord,	$O(\log n)$	$O(\log n)$	1, finite	Hashing	High	No
HyperCup	$O(\log n)$	$O(\log n)$	1, finite	Hashing	High	No
Pastry	$O(\log n)$	$O(\log n)$	1, finite	Hashing	High	Med
Tapestry	$O(\log n)$	$O(\log n)$	1, finite	Hashing	High	Med
Kademlia	$O(\log n)$	$O(\log n)$	1, finite	Hashing	High	Med
DHash	$O(\log n)$	$O(\log n)$	1, finite	Hashing	High	Med
Bamboo	$O(\log n)$	$O(\log n)$	1, finite	Hashing	High	Med
P-Grid	$O(\log n)$	$O(\log n)$	1, infinite	Trie-Mapping	High	Med
SkipNet	$O(\log n)$	$O(\log n)$	1, infinite	None	High	No
Chord#	$O(\log n)$	$O(\log n)$	1, infinite	None	High	No
Papnet	$O(\log n)$	$O(\log n)$	1, infinite	None	High	High

Figure 2.1: Comparison of different P2P Overlay Networks

However, the utilized gossiping protocols require communication overhead of $O(n)$ with each gossip message exchange, which is fine for small clusters of tenth to hundreds of nodes, but is certainly not applicable in true large-scale systems (e.g. today's peer-to-peer file-sharing systems with millions of participating nodes [43]). Assuming a node state descriptor of $\approx$ 20 Bytes and a system size of 1,000,000 nodes, each gossip message would comprise $\approx$ 20 Megabytes of data. Nonetheless, most of the P2P systems being in production use belong to this class of Overlays. Examples are Cassandra [17], Dynamo [16] and Riak [44]. Up to now, the largest reported cluster size of the most prominent representative Cassandra is 400 nodes[5]. Since linear node degree networks are essentially not scalable to large node counts, we neglect them in the context of this thesis.

Constant Degree Networks

Networks with constant node degree provide the advantage, that the overhead induced by peer-link maintenance is independent from the actual network size. A major disadvantage is the lack of link redundancy. Failures of nodes can cause temporary unavailability and since certain links are used more frequent than others for messages routing, severe node overloads may arise.

One of the first constant-degree networks was CAN (*Content-Addressable Network*) [36], which provides a d-dimensional continuous key space, e.g. a 2-dimensional

[5]http://gigaom.com/cloud/cassandra-snags-u-s-government-runs-on-amazon-ec2/

torus surface. It requires the maintenance of only $2 \cdot d$ links to other peers, but also provides only a sub-optimal routing path length of $O(\sqrt[d]{n})$. CAN allows for *Proximity Route Selection* (PRS) [45], which means that low latency links may be preferred when routing a message, while the actual routing link establishment is deterministic. However, this kind of latency-awareness is limited by the lack of choice due to the constant number of outbound links. For load balancing reasons, CAN requires the usage of a hash-function so that nodes and objects can be assumed to be distributed uniformly in key space. This effectively limits keys to be of finite length and also prevents range queries.
Viceroy [46] and Koorde [47] are further examples for constant degree networks and they even enable an efficient routing complexity of $O(\log n)$. Viceroy is based on a Butterfly Network [48], while Koorde implements a DeBruijn Graph [49].

Logarithmic Degree Networks

Networks with logarithmic node degree introduce redundancy and thus robustness and fault tolerance into the overlay topology. There are basically two sub-classes of such networks: Those that require a special key mapping (e.g. Hash-function) or make assumptions about the availability of initial sample data, and those that are able to handle arbitrary data without initial or runtime assumptions.

Logarithmic Degree Networks with Mapping Requirements

Chord [2] is the most prominent and also most simple representative of a P2P network with logarithmic node degree. Each node establishes links, so called *Fingers*, to $O(\log n)$ other nodes in exponentially increasing distances in key space. A uniform distribution of nodes in the key space is assumed (and achieved by requiring a key mapping via a Hash function). Further, each node maintains links to its direct successors in key space, which provides additional robustness.

HyperCup [50] is a network based on a Hypercube topology, where each node maintains links in $\log n$ (Hypercube-)dimensions. Both HyperCup and Chord implement a 1-dimensional finite key space, but depend on assumptions about the key distribution and therefore require the utilization of Hash-function mappings. Neither one provides means to minimize routing latency.

Pastry [3], Tapestry [4] and Kademlia [19] all define topologies that enable prefix-oriented routing using digit-by-digit target key matching. The requirements on the peer links are loose, so that peers are able to choose proximate nodes with regards to latency as routing links. This flexibility is called *Proximity Neighbor Selection*

(PNS) [45] and provides better locality than PRS. In section 2.4 we will describe the representative network *Pastry* in more detail and will provide the reader with more insight into prefix-oriented routing.

P-Grid [51] is a network that also enables prefix-oriented routing by implementing a virtual binary search tree over the key space. In contrast to the aforementioned networks, P-Grid allows keys to be stored in a sorted manner, but requires an order-preserving mapping of keys using a structure called *Trie*. The latter is assumed to be constructed based on an initial training data set, which therefore needs to be available. Other more advanced networks that do not require such initial assumptions will be covered in the next section.

While all prefix-oriented networks allow for latency optimization through PNS, they do not *guarantee* that any node will ever learn about its most proximate neighbors. Links are optimized passively by measuring round trip times each time a message from a yet unknown peer is received. In contrast, the network presented in this thesis – Papnet – is able to *guarantee* fast eventual convergence against the most proximate links by introducing a unique active optimization technique that induces only an efficiently small overhead (see Chapter 5).

Assumption-free Logarithmic Degree Networks

This class of networks does not make any initial assumptions on the data set and does not require a transformation of keys. Networks of this class define their overlay topology independent of the actual key space and can thus store keys in a sorted manner and allow for range queries. An example network is *SkipNet* [5], which constructs efficient routing links based on a probabilistic schema. SkipNet will be covered in detail in section 2.5.

Chord# [6] is an extension to Chord that explicitly constructs routing links in exponentially distances in node space instead of the key space. The latter is achieved by constructing the k^{th} Finger of a node by asking the $(k-1)^{\text{th}}$ Finger for its own $(k-1)^{\text{th}}$ Finger. This cooperative construction of exponentially increasing links effectively enables a decoupling of network topology and key space.

Papnet is the network presented in this thesis. It uses the same deterministic construction schema for its routing links as Chord#, but has been developed independently. There are, however, key differences: Papnet provides a latency-aware routing and implements bi-directional links, which enable an active optimization techniques that guarantees an eventually optimal routing latency.

2.4 Pastry

Pastry was one of the first structured second generation overlays. The topology and routing algorithm of Pastry are based on Plaxton's Algorithm [52], a prefix-based routing method that is also used in the *Tapestry* [4] Overlay. Pastry was proposed as early as 2001 and provides a good introduction into P2P Overlay networks. Although the Chord Overlay [2] is often preferred as an introduction to P2P Overlays because of its simplicity, we chose Pastry because we believe it to be very simple as well and because of its strong relations to Papnet, the new Overlay we present in this thesis. The detailed introduction of Pastry will provide the reader with background information and understanding that will be useful when reading the later chapters.

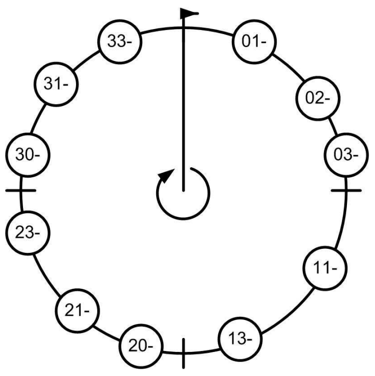
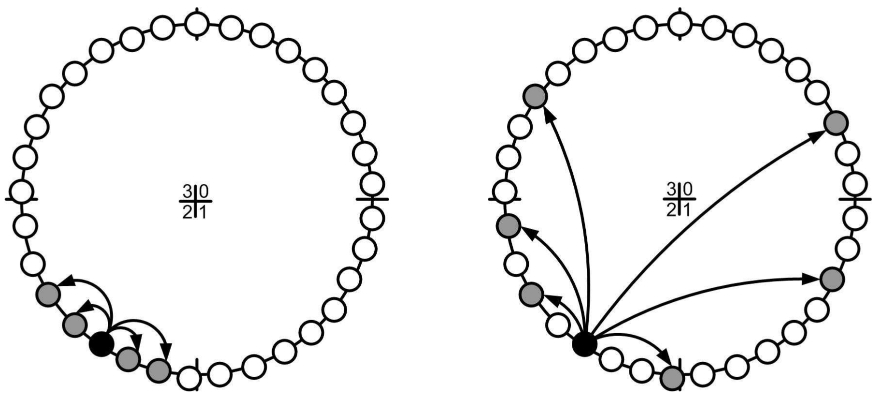


Figure 2.2: Pastry Key Space, $b = 2$.

Pastry defines a circular key space of cardinality 2^{128}. Each participating node is assigned a random 128 bit id in the key space. All keys represent sequences of $\frac{128}{b}$ digits from a digit set of cardinality 2^b. b is a global constant that needs to be a factor of 128. Implementations of Pastry often set $b = 4$, so that the digit set is the hexadecimal numbers. Figure 2.2 shows an exemplary key space where $b = 2$.

Leafset and Routing Table

Each Pastry node manages a set of pointers to known neighbors, called the *leafset*. This set has cardinality l and its components point to the $\frac{l}{2}$ numerical closest neighbor nodes in both directions along the circular key space. Leafsets ensure the



(a) Leafset (b) Routing Table

Figure 2.3: Pastry Node State.

basic connectivity of the overlay network and guarantee termination when routing messages. Figure 2.3a illustrates a leafset of cardinality $l = 4$.

To provide efficient routing towards arbitrary keys, each node also maintains a local *Routing Table* containing pointers to other nodes in exponentially increasing key space distances. Essentially, these pointers allow to halve the numerical distance to any target key in each routing step. Figure 2.3b visualizes the routing table entries in the circular key space.

The routing tables consist of $\frac{128}{b}$ rows and 2^b columns of pointers to other nodes with each entry satisfying two properties:

1. An entry in row j has a prefix of exactly j digits with the local node's ID.
2. The $(j + 1)$th digit of the ID of an entry in row j and column i is i.

Figure 2.4 shows an example routing table of a pastry node in network where $b = 2$ and the key bitlength is 8, i.e. keys are strings of 4 digits.

j \ i	0	1	2	3
0	**0**102	**1**321	–	**3**113
1	–	*2* **1**31	*2* **2**02	*2* **3**33
2	*20* **0**1	–	*20* **2**3	*20* **3**1
3	*201* **0**	*201* **1**	*201* **2**	–

Figure 2.4: Routing table of node **2013**.

Routing

Pastry employs a prefix-oriented message forwarding to enable an efficient routing towards arbitrary addresses. The asymptotic number of forwardings required to reach arbitrary targets is in $\mathcal{O}(\log_{2^b} n)$ [3]. This is accomplished by continuously forwarding messages to nodes having a longer common prefix with the destination.

The routing table provides a row for any given prefix length p, $0 \leq p < \frac{128}{b}$ that contains nodes that share a common prefix of length p with the local node id. This row contains one column with a node that has a common (p+1)th digit with the target address and whose prefix with the target address is thus one digit longer. If the corresponding table entry is empty, Pastry uses the leafset to perform numerical routing to a node that shares at least an equally long prefix with the target key, but is numerically closer to the target. If the target key is contained in the key space range covered by the local leafset, the above routing is skipped and the message is being forwarded to the responsible leaf node, which is the node numerically closest to the target key. Figure 2.5 shows an example routing.

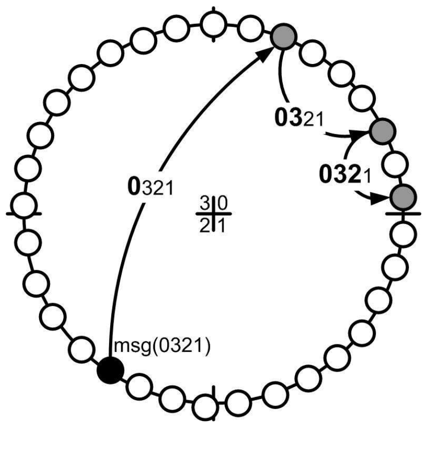


Figure 2.5: Pastry message routing example.

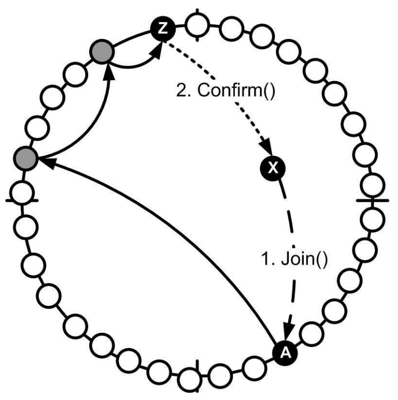
Pastry always delivers messages to the node that is globally most numerically close to the target id. Thus each pastry node is responsible for an exclusive partition of the global key space, which is defined by its own id and the ids of its directly neighboring nodes. Algorithm 1 shows the pseudocode of the routing algorithm, where D is the target id, A the id of the current node, L the current nodes leafset, L_-, L_+ the most numerically distant nodes in the leafset and $R_{j,i}$ the routing table entry in row j, column i.

Algorithm 1: Route(D, msg)

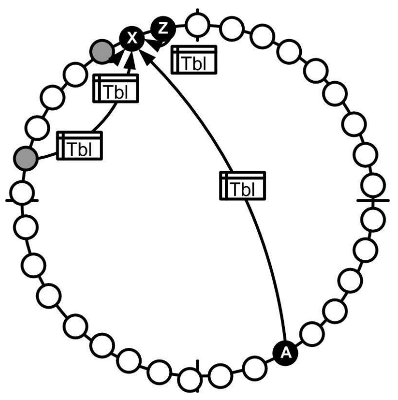
```
1  if $D \in [L_{-}..L_{+}]$ then
2      // D lies within the leafset range
3      Forward msg to $L_i \in L$, where $|L_i - D|$ is minimal.
4  else
5      // Use Routing Table
6      Let $p =$ Length of shared digit prefix of $A$ and $D$
7      Let $z =$ the $(p + 1)$th digit of $D$
8      if $R_{p,z} = null$ then
9          Forward msg to the unique node in $\{A\} \cup R \cup L$, that shares at least
           a common prefix of length $p$ with $D$ and is numerically closest to $D$.
10     else Forward mgs to node $R_{p,z}$.
```

Bootstrapping a new node

A new node at first chooses a unique id X, e.g. randomly or by calculating the SHA-1 hash [11] of its IP address. Then, it contacts an arbitrary known node A that is already a member of the network. A will then send a special *join* message towards the key X, which will get routed like a normal Pastry message and delivered on the node Z whose ID is closest to X. Z will then acknowledge the join request to X. In case both X and Z are equal, Z will deny the join request and X will need to choose a new ID and restart the join process. Figure 2.6a illustrates the join sequence.



(a) Join Sequence



(b) Routing Table Construction

Figure 2.6: Bootstrapping in Pastry.

Each node that is visited by the join request sends its routing table to the new node X (as depicted in Figure 2.6b). Using this information, X is able to construct its own initial routing table. The last node on the join path provides an initial leafset to X. As soon as X has successfully initialized its table, it notifies all known other node about its arrival. These nodes will in turn update their own tables.

Node Failure

Pastry periodically sends *heartbeat* messages to detect failed leafset nodes. In case a node failure has been detected, the leafset will be filled up by asking the numerically most distant node in the local leafset for its own leafset. Note that heartbeat messages are only exchanged between leafset nodes. Failures of nodes in the routing table will be detected *on the fly*, i.e. when a message routing using a table link does not get acknowledged. To repair the routing table entry, candidates will be requested from the other entries of the same routing table row j. In case there is no such node, entries in rows $j' > j$ will be asked for candidates. Using this strategy, it is highly likely that a replacement node will be found.

Latency Awareness

The prefix oriented routing allows Pastry to optimize the routing table entries for proximity, e.g. lowest latency. Since node IDs are assigned randomly and the row number corresponds to the length of the common prefix, there are on average $\frac{n}{b}$ candidates for each of the b columns in row 0. More generally, the expected number of candidates $C(r)$ for a particular entry in row r is

$$C(r) = \frac{n}{b^{r+1}}.$$

Let ρ denote the mean latency between any two nodes. Assuming a uniform distribution of latencies, we can expect half of the candidates $C(r)$ to have a latency lower than ρ. Further, we can expect that half of this half candidates do have a lower latency than $\frac{\rho}{2}$. In general, this extends to $\frac{C(r)}{2^k}$ candidates having a latency lower than $\frac{\rho}{2^{k-1}}$ after k halvings.

Thus, the routing steps using the lower-prefix routing table rows – those where the candidate sets are large – will only induce exponentially decreasing latency costs. Considering a typical routing path, the average total routing latency thus amortizes to a constant factor of the mean latency. In particular, the lowest possible path latency is $2 \cdot \rho$. A more detailed derivation is given in section 4.3 of [27].

2.5 SkipNet

In contrast to Pastry, *SkipNet* by Harvey et al. [5] is a P2P Overlay that does not require any hashing of object/node keys. It is thus able to store objects in a sorted manner and also allows for range and wildcard queries. SkipNet is based on the concept of *Skip Graphs*, which are themselves based on *Skip Lists* [53]. In the following, we will describe these three structures.

Skip List

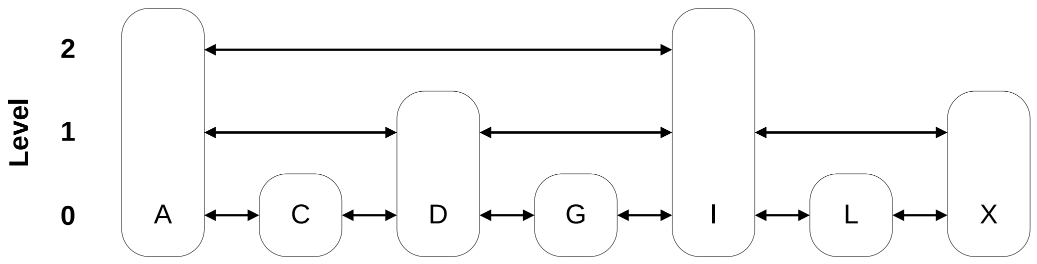


Figure 2.7: Skip List

Figure 2.7 shows an example Skip List of nodes. On the bottom, one can see that a Skip List is based on a classic linked list. In order to make lookups (as well as inserts and deletions) more efficient, a Skip List constructs multiple additional levels of lists that only contain subsets of nodes. These additional lists can be used to *skip* ranges of nodes and thus speed up the lookups of keys.

Each node in a Skip List is part of the linked list at level 0 and in a number of higher levels. In particular, a node is a member of the list at level l with a fixed probability p, given that it is also part of level $l-1$. The total probability of a node being a member of the list at level x thus calculates to p^l. The higher levels of lists represent "Express Lanes" which skip numbers of nodes that grow exponentially with increasing level. These lists therefore enable fast lookups which result in logarithmic search time complexity. In particular, the expected average search time is in $O(\log n \frac{1}{(1-p)\log\frac{1}{p}})$ [54].

Skip Lists are however not very well suited to be used in a distributed system, because the nodes having links at higher levels tend to get overloaded and their failure can have severe impacts on the routing performance due to the lack of redundancy. In [54] the authors discuss the above disadvantages and propose a new structure called *Skip Graph*.

Skip Graph

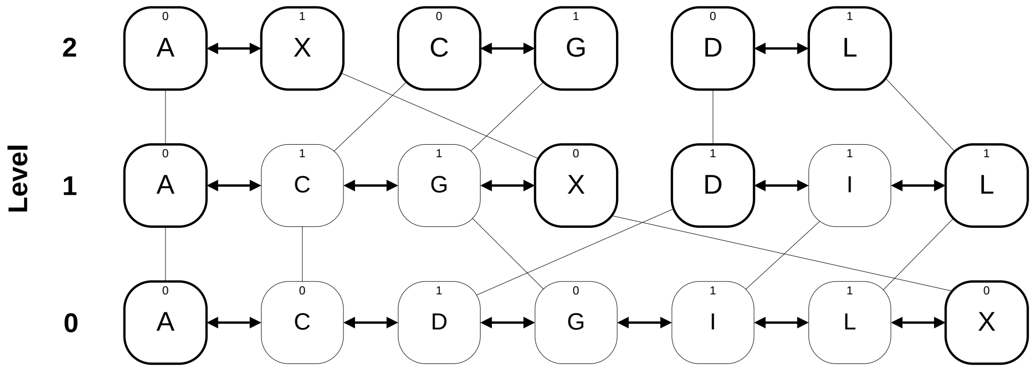


Figure 2.8: Skip Graph

A *Skip Graph* [54] extends the concept of Skip Lists by introducing multiple lists per level. In contrast to Skip Lists, *all* nodes are part of *some* list at each level, except for the basic list at level 0, which is unique. The list memberships for any node x are defined by a (infinite) random membership vector $m(x)$ over a finite alphabet, e.g. binary values. The different lists at some level l are identified by a unique finite string w of length l over the same alphabet. At level l, a node x is part of the list w, iff $|w| = l$ and w is a finite prefix of $m(x)$.

Figure 2.8 depicts an example Skip Graph of 7 nodes. There are six lists: the base list that contains all nodes, two lists at level 1 and three at level 2. Lists that only consist of a single node are not shown (i.e. the list that only contains node I at level 2). Note that all nodes in a list at some level l are also in common lists at all levels lower than l, since they share a common membership prefix. The (finite) membership vectors for the depicted scenario are: $m(A) = 000$, $m(C) = 010$, $m(D) = 110$, $m(G) = 011$, $m(I) = 101$, $m(L) = 111$ and $m(X) = 001$.

In practice, each node in a network of size n is only part of $O(\log n)$ lists, because higher list levels do not contain any other nodes and can therefore be neglected. The routing in a Skip Graph is performed exactly as in Skip Lists, with the exception that different lists are used as "Express Lanes" at higher levels. A Skip Graph thus introduces redundancy and fault tolerance to Skip Lists, while preserving a routing complexity of $O(\log n)$ and keeping the state of each node efficiently small ($O(\log n)$ links).

SkipNet

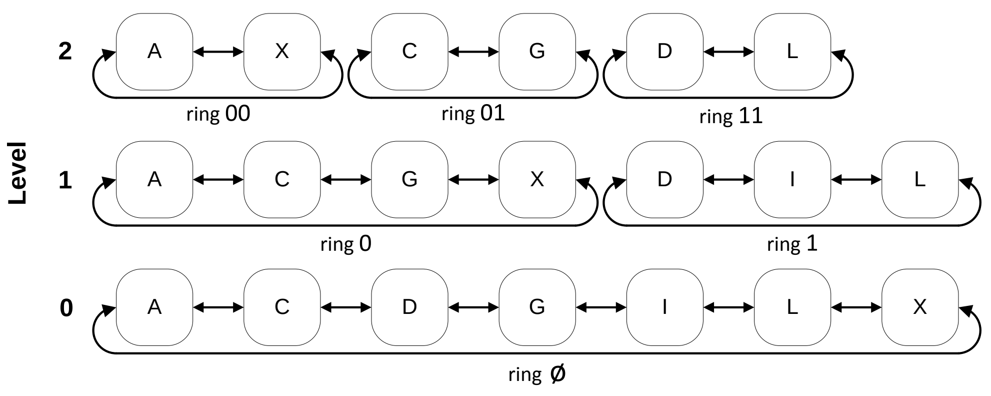


Figure 2.9: SkipNet

SkipNet implements a distributed Skip Graph structure according to the description given in the last section, with the difference that all lists are extended to wrapping ring structures (i.e. the last node in each list has a pointer to the first node and vice versa). Each node has a so-called *Name ID* that determines its position in each ring that it is a member of. The ring memberships of a node are defined by a binary string of fixed length, which is called the *Numerical ID*.

In addition to the routing by Name IDs in $O(\log n)$, SkipNet also enables an $O(\log n)$ routing by Numerical ID. This is achieved by simply following the links of a ring in one direction until a node with a larger common prefix with the target ID is found. The algorithm starts at the level 0 ring and each time such a node with larger prefix is found, the algorithm will continue with the next higher level. Since all nodes in a ring at level l share a common prefix of length l and the numerical IDs are chosen randomly, the probability that the (next) node has a common prefix of at least l+1 with the target is 50% (using a binary alphabet). Therefore, it is highly probable to reach any Numerical target ID with $O(\log n)$ routing steps.

While SkipNet is able to handle any distribution of Name IDs and thus does not require any normalization of IDs through a Hash function mapping, it does not support Proximity Neighbor Selection (PNS) [55]. PNS is the ability to prefer proximate neighbors as routing links to minimize the routing latency and can be applied to classic DHTs such as Pastry, but not to SkipNet. This is because the static Numerical IDs of SkipNet nodes need to be chosen randomly and the derived ring memberships lead to a strongly deterministic establishment of routing links.

2.6 Ganesan Online Load Balancing

The *Online Load Balancing of range-partitioned Data* proposed by Ganesan/Bawa/ Garcia-Molina in [7] is a load balancing algorithm that operates on range-partitioned data. The core algorithm only requires local knowledge and is thus suitable to be implemented on top of a P2P overlay.

Model

Consider a relation that has been partitioned into n segments, that are defined by a set of $n+1$ (or n, in case of a wrapping domain) range boundaries $\mathcal{R}_0 \leq \mathcal{R}_1 \leq \cdots \leq \mathcal{R}_n$. Each range $[\mathcal{R}_{i-1}, \mathcal{R}_i)$ is assigned to a single node N_i in the distributed system. The special case $\mathcal{R}_{i-1} = \mathcal{R}_i$ is called an *empty range*. The *load* of a node N_i is denoted by $L(N_i)$ and represents e.g. the number of objects stored in its responsibility range $[\mathcal{R}_{i-1}, \mathcal{R}_i)$.

Load Thresholds

As objects are inserted and deleted, the loads of the nodes change. Whenever major imbalances arise in the network, load balancing operations need to be performed. The algorithm defines a series of thresholds $T_{i\in\mathbb{N}}$ that are used to trigger the execution of such operations, whenever they are crossed. The thresholds are defined based on a global parameter δ and constant c:

$$T_{i\in\mathbb{N}} = \lfloor c\delta^i \rfloor.$$

The lowest possible value of δ is the golden ratio $(\sqrt{5}+1)/2 \simeq 1.62$ [7]. In case $\delta = 2$ and $c = 1$, the thresholds are the powers of two:

$$\bigcup_{i\in\mathbb{N}} T_i = \{1, 2, 4, 8, ...\}.$$

The Ganesan Online Load Balancing algorithm defines exactly 2 operations that utilize these load thresholds: *NeighborAdjust* and *Reorder*.

Operation NeighborAdjust:
Two adjacent nodes A and B that have a load difference of at least 1 threshold perform a load balancing by transferring data objects and readjusting the range boundary in-between them. Figure 2.10 visualizes this operation. Note, that this is a purely local operation that only involves these two neighboring nodes.

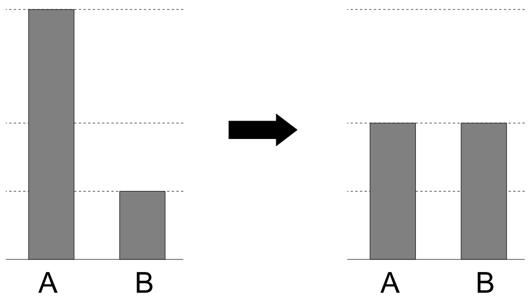


Figure 2.10: *NeighborAdjust*

Operation Reorder:
A low loaded node C that has at least 2 thresholds less load that an overloaded node A transfers all its data to one of its neighboring nodes so that its partition becomes empty. Then, it changes its position to become a neighbor of A, splitting its responsibility range $[\mathcal{R}_{A-1}, \mathcal{R}_A)$ at some key X within that range, so that C becomes responsible for $[\mathcal{R}_{A-1}, X)$, while node A is now responsible for a smaller range $[X, \mathcal{R}_A)$. Figure 2.11 illustrates this operation.

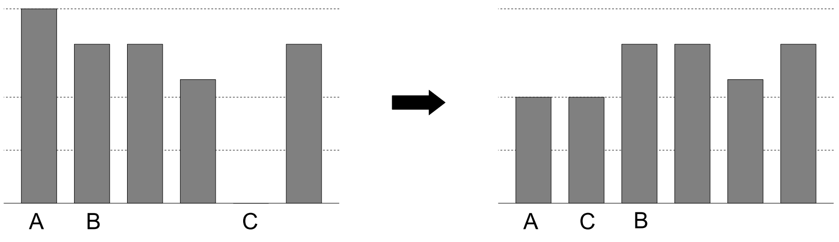


Figure 2.11: *Reorder()*

Local Knowledge

In order to execute the *Reorder* operation, the globally least loaded node needs to be determined. In a distributed P2P setting this is a challenging task, since each node only has partial local knowledge about the network. However, in [7] the authors also present a randomized version of the algorithm, that only requires a sample set of $\log n$ nodes to pick the least loaded node from. They show that the randomization only slightly influences the load distribution.

Load Balancing

Each time the insertion of an object causes the load of a node N_i to rise to a value of $L(N_i) = T_m + 1$ for some m, a load balancing is triggered by executing *AdjustLoad*. If the lighter loaded of the directly neighboring nodes N_j has a load $L(N_j) \leq T_{m-1}$, then N_i will transfer some of its load to N_j, using the *NeighborAdjust()* operation to equalize both nodes loads. In case both neighbors have a higher load, a global balancing operation becomes necessary by looking up the least loaded node in the network. This node will then execute the *Reorder()* operation, i.e. transfer its local load to one of its own neighbors, relocate to become a neighbor of N_i and take over half of its load. Algorithm 2 presents the corresponding pseudocode.

Algorithm 2: AdjustLoad(N_i)

```
1  Let $L(N_i) = x \in (T_m, T_{m+1}]$
2  Let $N_j$ the lighter loaded neighbor $N_{i-1}$ or $N_{i+1}$ of $N_i$
3  if $L(N_j) \leq T_{m-1}$ then
4      Transfer data from $N_i$ to $N_j$
5      AdjustLoad($N_j$)
6      AdjustLoad($N_i$)
7  else
8      Lookup the globally least loaded node $N_k$
9      if $L(N_k) \leq T_{m-2}$ then
10         Transfer all data from $N_k$ to $N = N_{k\pm1}$
11         Transfer data from $N_i$ to $N_k$, so that $L(N_i) = \lceil \frac{x}{2} \rceil$, $L(N_k) = \lfloor \frac{x}{2} \rfloor$.
12         AdjustLoad(N)
13         Rename nodes appropriately
```

Imbalance guarantees

The presented algorithm provides an upper constant bound on the global load imbalance ratio σ, which depends on the chosen threshold base δ and calculates to $\sigma = \delta^3$. The total load imbalance invariant is defined by:

$$\forall N_{i,j}, L(N_i) > L(N_j) : L(N_i) \leq \sigma \cdot L(N_j) + c_0, \text{ for some constant } c_0.$$

Thus if $\delta = 2$ the maximum imbalance is $\sigma = 8$, which means that the most loaded node has at most 8 times as much load as the least loaded node. The lowest possible imbalance ratio is ≈ 4.236 using the golden ratio as threshold base: $\delta = (\sqrt{5} + 1)/2 \simeq 1.62$. Further, the load balancing algorithm has only constant amortized costs for insertion and deletion of data objects.

2.7 Distributed RDF Stores

A mostly academic application scenario for P2P Overlays is the realization of a distributed RDF store. RDF stands for *Resource Description Framework* and can be used to define and reason about ontologies [56]. An RDF graph can be entirely expressed by so-called *Triples*. These consist of 3 three attributes called *Subject*, *Predicate* and *Object* (S, P, O) and describe relations between entities in the RDF-Graph, e.g.

$$(\textit{Stop Signal},\ \textit{blinks},\ \textit{Red})$$
or
$$(\textit{Red},\ \textit{is},\ \textit{Color}).$$

A typical query in an RDF store requires joins of triple sets that are each described by only two attributes. An example query for the *Color of a blinking Stop Signal* could be expressed as

$$(\textit{Stop Signal},\ \textit{blinks},\ \textit{?x}) \wedge (\textit{?x},\ \textit{is},\ \textit{Color})$$
or more generally
$$(S_1,\ P_1,\ \textit{?x}) \wedge (\textit{?x},\ P_2,\ O_2)$$

Distributed RDF stores enable the processing of such a query by storing each triple three times at keys calculated from two attributes, e.g. at $h(S \cdot P)$, $h(P \cdot O)$ and $h(O \cdot S)$, where h is a hash function.
Given an example query with two known attributes $(S,\ \textit{?},\ O)$ all matching candidates can be found at key $h(O \cdot S)$. However, popular triples such as $(\textit{*},\ \textit{is},\ \textit{Red})$ do create load hotspots. When using a classic DHT, all such triples need to be stored at the very same node, which inevitably leads to load imbalances. Example RDF stores using the DHT approach are RDFPeers [57] and Babelpeers [58].

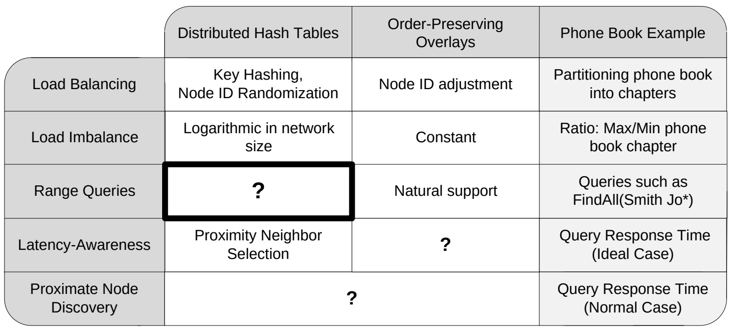
Using an order-preserving Overlay, the DHT-strategy can be refined to avoid hotspots and distribute load evenly. Still, all triples will be stored at three different keys

$$\begin{aligned} k_1 &= S \cdot P \cdot O, \\ k_2 &= P \cdot O \cdot S, \\ k_3 &= O \cdot S \cdot P, \end{aligned}$$

but now all such keys are unique. All candidates for a query like $(\textit{?},\ P,\ O)$ can be found in a consecutive range of keys, starting at key $P \cdot O$. This application motivated the development of an extension to the classic Pastry Overlay that allows for a continuous ID space, which eventually lead to the invention of Papnet.

3 Enabling DHT Range Queries

Contents



	Distributed Hash Tables	Order-Preserving Overlays	Phone Book Example
Load Balancing	Key Hashing, Node ID Randomization	Node ID adjustment	Partitioning phone book into chapters
Load Imbalance	Logarithmic in network size	Constant	Ratio: Max/Min phone book chapter
Range Queries	**?**	Natural support	Queries such as FindAll(Smith Jo*)
Latency-Awareness	Proximity Neighbor Selection	**?**	Query Response Time (Ideal Case)
Proximate Node Discovery	**?**		Query Response Time (Normal Case)

Figure 3.1: Overlay Properties

3.1 Introduction

In this chapter, we present the idea to combine the advantages of Pastry including locality and both low routing and space complexity of $\mathcal{O}(\log n)$ with the advantages P2P networks that store objects based on a total order, e.g. SkipNet [5]. To answer the question of how to enable range queries in classic DHTs (as depicted in Figure 3.1), we introduce an alphanumerical id space over Pastry's numeric id space. Each peer is assigned a second identifier from the alphanumerical id space: its name. Thus, besides the standard routing using hashed ids, the network is also able to route using these name ids and thus able to perform range queries.

Further, an alphanumeric overlay allows for the implementation of *Ganesan Online Load Balancing* (GLB) [7], since it provides means to dynamically change the responsibility between two neighboring nodes to balance the load. We show how our extension can be used to apply GLB and which limitations do arise.

The remainder of the chapter is structured as follows: Section 3.2 gives an overview about the related work. In Section 3.3 we introduce the alphanumerical ids including the changes in Pastry that are necessary for routing. The next section explains how load balancing can be performed and discusses advantages and disadvantages. The performance of this new routing possibility is examined in Section 3.5 and Section 3.6 concludes the chapter.

3.2 Related Work

One special goal of our approach is to apply Ganesan Online Load Balancing [7], a load balancing technique that is able to guarantee a constant load imbalance ratio. Up to now, this algorithm cannot be applied to most Overlay networks.

Classic DHTs assume a uniform distribution of both nodes and data in the ID space and try to equalize storage load by optimizing imbalances in the responsibility ranges of nodes. One possibility is the introduction of *Virtual Servers* [59], which is the concept of running multiple overlay nodes per physical node. Since each real node's responsibility becomes the sum of all its overlay nodes' ranges, the overall imbalance decreases. The approach has been used by Ledlie and Seltzer in the context of their *k-choices* algorithm [60]. However, it requires a maintenance overhead proportional to the number of virtual instances that each node runs.

Another way is the usage of "Power of two" strategies as suggested in [61] or [62, 59, 63]. Here, the key idea is to use up to k hash functions instead of a single one. Other DHT-specific load balancing strategies e.g. those proposed in [62, 59, 63] attempt to repartition the ID space among nodes. All strategies mentioned so far do however only work under the assumption of uniform distribution of data keys in the ID space, but in case data is skewed, an equalization of responsibility ranges helps little to balance the load distribution.

SkipNet [5] is an overlay network that provides a numeric as well as a dynamic alphanumeric ID space and is able to process efficient range queries. Further, it allows for the implementation of GLB. The SkipNet structure, however, lacks useful locality properties provided by other overlay networks such as Pastry [3], since routing links are established strongly deterministic and symmetric.

3.3 Alphanumerical Extension

Before describing the extensions for alphanumerical routing, we introduce some definitions. First of all we need a globally known arbitrary but fixed **character set** denoted by $\mathcal{C}$. We use $\bot$, $\top$ to denote the minimum and maximum character in $\mathcal{C}$, so that $\forall c \in \mathcal{C} : \bot \leq c \leq \top$.

Now, we define the set of all possible **alphanumerical ids** (names) by $\mathcal{K} := \mathcal{C}^+$ as sequences of characters. Hence, a **nameID** $w \in \mathcal{K}$ is defined as $c_1 \cdot c_2 \cdot c_3 \cdots c_k$ with $c_{1..k} \in \mathcal{C}$ with length $|w| := k$.

To enable sorting of all name ids, we will now introduce a relation $\prec$ that defines a **total order** on all elements in $\mathcal{K}$. At first, we demand that $\forall w \in \mathcal{K} : w = w \cdot \bot^+$, which means that we are allowed to append arbitrary numbers of $\bot$ to any name.

Now that we can treat any two names $a, b \in \mathcal{K}$ as if they had equal character length, we define $a \prec b$, iff $|a| = |b|$ and a is smaller than b according to a classic lexicographical order. $\preceq$, $\succ$, $\succeq$ are defined analogously. Note that $\forall a, b \in \mathcal{K}$ and w.l.o.g. $a \prec b$ we can construct a new nameID c for that holds $a \prec c \prec b$.

At last we define the **alphanumerical distance**. Let $a, b \in \mathcal{K}$ and w.l.o.g. $|a| \geq |b|$ then the alphanumerical distance $|a - b|$ is defined as the numerical distance between a and the id of equal length b', which is constructed by appending $\bot$ digits to b: $b' = b \cdot \bot^{|a|-|b|}$.

With these definitions, we can introduce alphanumerical names for peers. Each node has a nameID beside the existing numerical id assigned through the Pastry network. The nameID of peer n is denoted as $name(n)$. It marks the end of the responsibility range $\mathcal{R}$ concerning key-value pairs stored in the alphanumerical overlay. Thus, peer n is responsible for the range $[name(pred(n)), name(n))$ with $pred(n)$ being defined as the predecessor of peer n (the counterclockwise neighbor in the Pastry ring). Just like the numerical ids, the node names increase along the ring, say $name(pred(n)) \prec name(n)$.

Now, we can define the **responsibility range** $\mathcal{R}_n$: Each node is responsible for all instances $w \in \mathcal{K}$ for which the following equation holds:

$$\mathcal{R}_n := \begin{cases} w \mid name(pred(n)) \preceq w \prec name(n) & \text{if } n \neq n_0 \\ w \mid w \prec name(n_0) \vee name(pred(n_0)) \preceq w & \text{if } n = n_0 \end{cases}$$

Using this definition $name(n_0)$ is not necessarily bigger than the biggest key stored on node n_0. It rather marks the end of the responsibility range for each peer. The set of objects which is actually stored on the node is denoted by $\mathcal{O}_n$. Thus $\mathcal{O}_n \subset \mathcal{R}_n$.

last(n) denotes the last existing key in the responsibility range of peer n. This means that no further element lies between *last*(n) and *name*(n). If $\mathcal{O}_n$ is empty, then *last*(n) is defined to be the name of the predecessor node.

$$last(n) := \begin{cases} o \in \mathcal{O}_n, \text{ so that } \forall o' \in \mathcal{O}_n : o' \leq o & \text{if } \mathcal{O}_n \neq \emptyset \\ name(pred(n)) & \text{if } \mathcal{O}_n = \emptyset \end{cases}$$

Figure 3.2 shows a cutout of an example alphanumerical ring around node n_0 including the responsibility range of peer n_0.

The nameID of a node is not fixed as a numerical id. Due to load balancing it is possible that the name ids of nodes are adjusted and therewith causing a shift of data from one node to its neighbor.

Join and Leave Algorithms

In most cases a P2P network is subject to continuous join and leave activities of peers. Pastry needs to handle this “churn” by executing a maintenance protocol that repairs dead links and ensures the coherence of the ring. To keep nameID information up to date this maintenance protocol is used to propagate the current names of peers. This section describes how a joining peer gets its initial name and what happens when a peer unexpectedly leaves the network.

As can be seen in Figure 3.2, there is always a gap between elements existing at two neighbored peers (*last*$(pred(n))$ and *name*$(pred(n))$): Because of the not limited length of alphanumerical ids and the total order, we can always generate

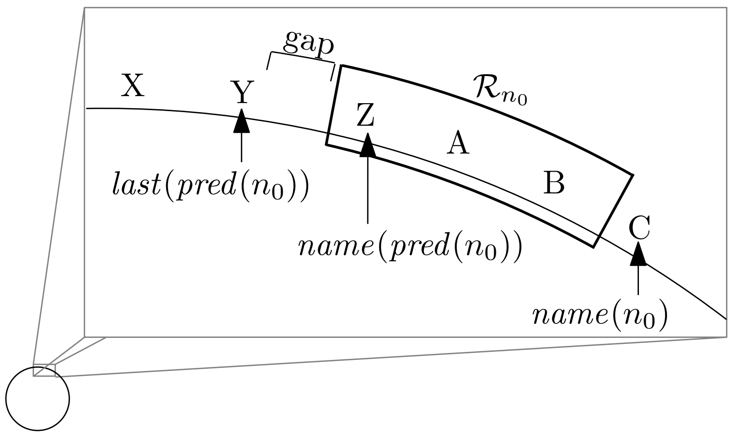


Figure 3.2: Responsibility Range of peer n_0

a nameID that fits into this gap. If e.g. $\mathit{last}(\mathit{pred}(n)) = Y$ and $\mathit{name}(\mathit{pred}(n)) = Z$ as depicted, then the gap is be defined to be the range (Y,Z) of all nameIDs between these two nameIDs which for example contains the fresh nameIDs YA.

When a new peer joins the network, it has no initial content but a fixed numerical id generated by Pastry. This numID determines the node's position in the ring. Standard join protocols are used to find neighbors and to fill the routing table with initial entries. The gap between the neighbors nameIDs is used to find a valid initial nameID matching the position on the ring.

Let n be the node joining the alphanumerical overlay. $\mathit{pred}(n)$ and $\mathit{succ}(n)$ are the direct neighbors of n. n sends a join message to $\mathit{pred}(n)$ and asks for its nameID. $\mathit{pred}(n)$ chooses a new name from the gap so that the responsibility range of $\mathit{pred}(n)$ gets a bit smaller but still comprises all existing instances $\mathcal{O}_{\mathit{pred}(n)}$. At last, n chooses the former nameID of $\mathit{pred}(n)$ as its own nameID. Using this algorithm, no movement of data is necessary. n gets responsible for some data subsequently via load balancing operations or new inserted elements that fall into its responsibility range.

Peers are allowed to leave the network at any time, without notifying other peers. The maintenance mechanisms of Pastry are responsible to repair the overlay in case of node departure and arrival of new nodes. Our extensions propagate newly introduced nameIDs as part of the node identifiers.

Routing

As described above, each routing information now also includes the nameID of the node besides its standard numerical id. Figure 3.3 shows an example for such routing entries in a Pastry network. Each node provides two data structures where routing information is stored, the leafset (dashed arrows) and the routing table (solid arrows).

The leafset of peer n stores information about nodes that are located in the neighborhood with respect to the position on the ring. In contrast, the routing table contains information about peers in other parts of the ring. How exactly Pastry finds these links is described in chapter 2.4. Note that Pastry has a certain degree of freedom when choosing peers for the routing table: it chooses the most proximate node that is located in a certain constrained area of the id space. Leafsets guarantee a better robustness of the network against unplanned node departures and do already enable a simple but slow alphanumeric routing algorithm: *Forward the message to that node in the leafset whose alphanumerical distance to the destination is the smallest.*

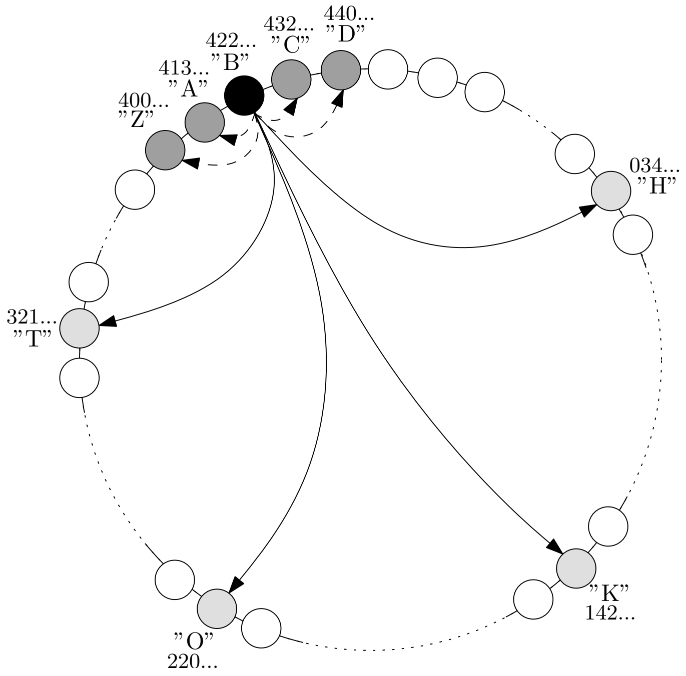


Figure 3.3: Example routing links of peer “B” (numID 422)

The routing algorithm for alphanumerical routing is given in Algorithm 3. Let D be the destination nameID and A the current node where the routing algorithm is performed. $succ(A)$ and $pred(A)$ are the clockwise and counterclockwise neighbors of A. R_{leaf} and R_{route} denote the set of nodes contained in the leafset, respectively routing table of node A. As long as nodes are distributed evenly on the ring, routing performance for the alphanumerical routing is the same as the DHT routing: $O(\log n)$. Effects of a load balancing algorithm on the routing performance are discussed in Sections 3.4 and 3.5.

Algorithm 3: Alphanumerical Id Routing

```
1 if $D \in [pred(A), A)$ then
2     Deliver message to myself
3 else if $D \in [A, succ(A))$ then
4     Route message to $succ(A)$
5 else
6     // Forward to closest node of routing links and leafset
7     Route message to node $Z \in R_{leaf} \cup R_{route}$, where $|Z - D|$ is minimal
```

3.4 Load Balancing

Load balancing is crucial for the performance in P2P networks. We can distinguish between storage and query load. Storage load refers to the number of elements a node is responsible for and query load to the amount of queries a node has to manage in a certain period of time. In DHT P2P networks, problems arise if data is not distributed evenly across the id space, or if there are many objects that map to the very same key. Heterogeneity of nodes introduces an additional parameter to this problem. Load detection is not in the focus of this paper but necessary for load balancing. In our simulations, we thus tested two different settings: Firstly, the (unrealistic) case where global knowledge of load information is used and secondly, the case of only local knowledge of nodes in the routing table.

The total order in the alphanumerical overlay enables a load balancing as presented by Ganesan et al. in [7] (GLB). A detailed description of the GLB is given in section 2.6. The algorithm defines two balancing operations: “NeighborAdjust” and “Reorder”. Load balancing is triggered when new data elements are inserted or deleted and a node crosses a load threshold or nodes arrive or leave the network. Firstly, an affected node tries a NeighborAdjust operation, and if this is not possible, it tries the more expensive Reorder operation. For determining whether a node is over- or underloaded, the algorithm uses globally known load thresholds defined as $T_i = \lfloor c\delta^i \rfloor$ with a constant c and a load imbalance parameter δ. Load balancing is triggered every time a node crosses a threshold. Throughout this chapter, we instantiate $\delta = 2$ and $c = 1$, so that the thresholds become powers of 2.

NeighborAdjust flattens load between neighbored nodes. If the load of a node is one threshold boundary higher than the load of a neighbor, the nodes move load by modifying their responsibility ranges through a change of a single node’s name id. After NeighborAdjust both nodes have the same load. Reorder balances load by removing and rejoining a low loaded node. It is applicable, if another node is known, whose load is at least two thresholds lower than its own load. The less loaded node m shifts all data to its neighbors, leaves the network and rejoins as a neighbor of the overloaded node n. Thus, it joins with a previously determined numID and not with a random numID. The rejoined node chooses the nameID X that is the element that splits the content of n exactly in the middle. If the node joins as predecessor (w.l.o.g.), its responsibility ranges become $\mathcal{R}_m = [name(pred(n)), X)$ and $\mathcal{R}_n = [X, name(n))$; $name(m) = X$. The numID must be from within the range between the numIDs of $pred(n)$ and n. As long as the distribution of newly inserted data keys is similar to the current distribution in the network, only few Reorder operations are necessary.

The *Reorder* operation affects the numerical Pastry overlay because of the determined numIDs. Whenever load imbalance is concentrated in a certain area of the id ring, a lot of nodes will get moved to this area by *Reorder* operations, causing a skewed distribution of nodes on the ring. With a growing skew, the load balance of the DHT suffers from load imbalance and possibly becomes not usable any longer. The numerous movements of peers into a small part of the numerical id ring can have several side effects. Firstly, the nodes are not longer distributed evenly on the ring, which is a basic assumption for routing performance estimations of $\mathcal{O}(\log n)$. Nevertheless, we assume that the actual effect on Pastry's numerical routing performance is rather small and analyze this in simulations. Results are presented in Section 3.5. Secondly, if load balancing causes reinsertion of nodes always as successors of peers, we soon encounter a lack of available numerical ids. This is discussed in the following.

Consider two nodes n and $pred(n) = m$. Now, the insertion of data causes a Reorder operation and moves a new node l between m and n exactly into the middle of both nodes' numIDs. Now a second Reorder operation again inserts a peer between $pred(n) = l$ and n, again exactly at the middle. Thus, the number of available numIDs between n and $pred(n)$ is reduced by half for every Reorder operation. Pastry has a limited number of numIDs, large enough to provide numIDs for a quasi arbitrary number of peers (if numIDs are chosen randomly). Usually, the size of the numerical id space is 2^{128}. It is obvious, that this id space can be halved only 128 times. This leads to only 128 possible Reorder operations like presented in the worst case above before ids run out. We can construct an example with 391 nodes and only 776 well directed insert operations to prevent the feasibility of further load balancing for a node in the network.

Through this fact, we conclude that the presented load balancing can cause problems in worst case. For the normal case, that can be constructed easily by inserting data in a random order, this situation never appeared during our simulations. Since we need to tackle this problem, we decided to modify the Pastry protocol (until now, we did only minor changes by propagating the name of nodes with the help of the maintenance protocols). We decided to change the finite numerical id space to an infinite id space because any restriction concerning the length of the numID will result in a restriction of the number of worst case Reorder operations. If there is no available numID for the node inserted through a Reorder operation, the reinserted node gets a numID that is one digit longer than the numIDs of the direct neighbors. The added digit is the middle element of the set of numerical id digits. Note, that by this change the numIDs become in fact nameIDs too (with exactly the same properties as defined in 3.3), but with a different character set. Nevertheless we still need both, static numIDs for the construction of the Pastry overlay and dynamic nameIDs for load balancing operations.

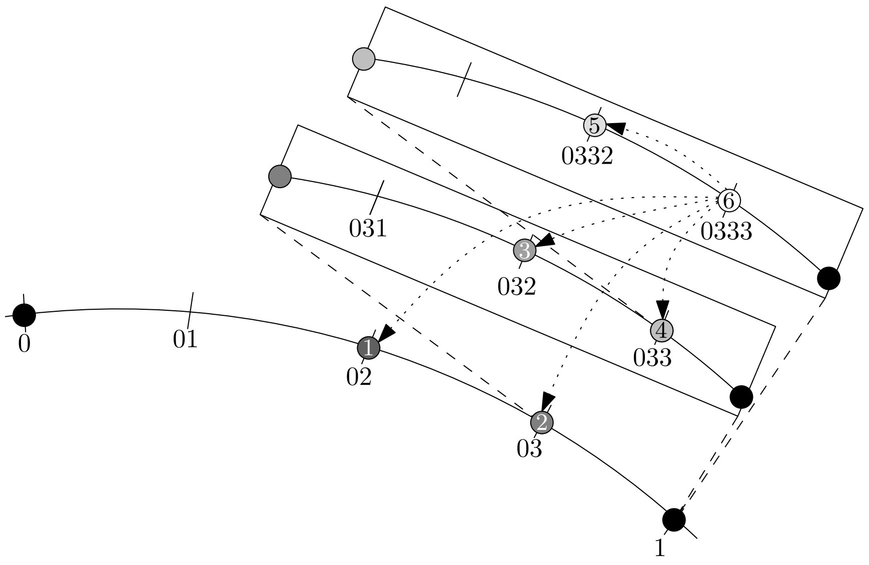


Figure 3.4: Densely Populated Area

Changes in the id space must be made very carefully and analyzed well, because these changes can have grave impacts on the routing performance. Note that Pastry's numerical routing algorithm is not directly affected by longer numIDs. The major problem is a possible growth in the size of routing tables, when multiple reorder operations lead to densely populated areas.

Figure 3.4 shows a fraction of an artificial example Pastry ring with $b = 2$ and (initial) numerical id length of 1 digit. There are already nodes at id 0 and 1 but load balancing requires more nodes between both. Therefore, nodes are inserted and their numIDs get extended. The numbers depicted inside the nodes denote the order of insertion. The numbers below the nodes are the numIDs. Furthermore, the links for node 0333 are shown for all nodes in the densely populated area.

A routing table with a high number of entries (larger than $\mathcal{O}(\log n)$) may produce higher maintenance costs as well. On the other hand, Pastry utilizes a lazy update mechanism to fix routing table entries, which only causes maintenance costs when routing fails. Due to the numID extension, it is hard to give exact estimations about the size of a routing table and the routing performance, because the proofs of Pastry's storage and routing efficiency are based on the finite address space and therewith an assessable size of the routing table. For evaluation of the impacts, we implemented a simulation of Pastry with the alphanumerical overlay including the described modifications.

3.5 Evaluation

We implemented a lightweight and idealized version of the pastry protocol to perform our simulations, that does neither simulate bootstrapping of nodes, nor their departure. The routing tables were generated using global knowledge of all existing nodes, setting each entry in a table to a random node picked from the set of matching nodes. This way, the tables got filled to their maximum, which greatly allows for worst case evaluation in terms of table size as well as best case evaluation concerning path lengths. The Pastry specific parameters used in all of the simulations were: $b = 4$ (hexadecimal set of digits), $|\text{leafset}| = 32$ and numerical IDs of (initial) length 128 Bit.

As scenario we simulate a distributed Resource Description Framework (RDF) database. RDF information are given through triples with **Subject (S)**, **Predicate (P)** and **Object (O)**. Each triple is one data unit and must be accessible if only one part of a triple is known. Several implementations of DHT RDF databases like RDFPeers [57], Babelpeers [58], and Atlas [64] exist and all distribute each triple at least three times by hashing each component (S, P, and O). RDF triples are connected through common Subjects and Objects to form a graph and some predicates with a special semantic occur very often. Thus, there are a lot of triples that are mapped to the very same hash value, which can cause a crucial imbalance that can not be flattened by a DHT.

When RDF data is stored in our alphanumerical overlay, each triple must be inserted in three different orders: Subject·Predicate·Object (SPO), Predicate·Object·Subject (POS) and Object·Subject·Predicate (OSP). Thus, each information is also accessible using range queries with wild cards for the unknown components if only one component is known.

The data set used in all of our simulations comprises an artificial RDF-graph generated by the Lehigh University Benchmark[65] (Lumb-10) that has been written to a file containing $1,272,764$ triples by Sesame 2.0[66]. This file has been used to create the actual list of data elements $(SPO_1, POS_1, OSP_1, SPO_2, ...)$, which results in a total of $3,818,292$ elements. The order of this list is what we call the *unaltered* Order of Data Inserts (OoDI). We define two additional orders which represent extreme cases: The *sorted* OoDI, where the data elements form an ascending list with respect to $\prec$ and the *randomized* OoDI, where the elements have been randomly permuted.

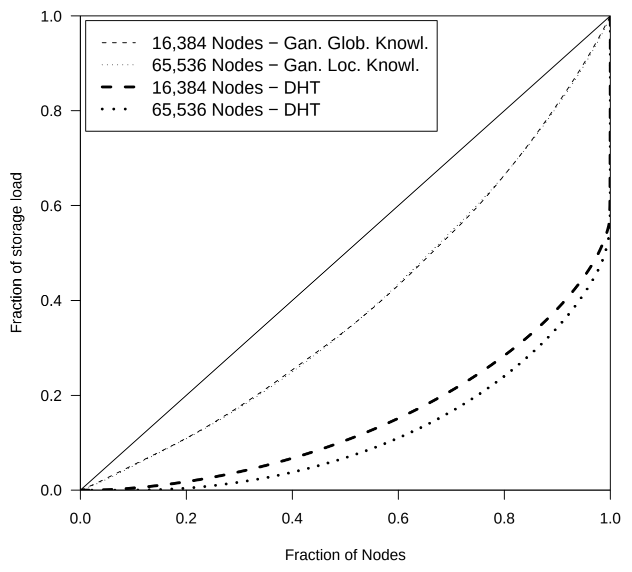


Figure 3.5: Lorenz-Curves of Load Distribution

Load Balance

To provide evidence for the necessity and efficiency of the alphanumerical overlay, we evaluated the performance of the Ganesan load balancing algorithm against a DHT in terms of the storage load balance.

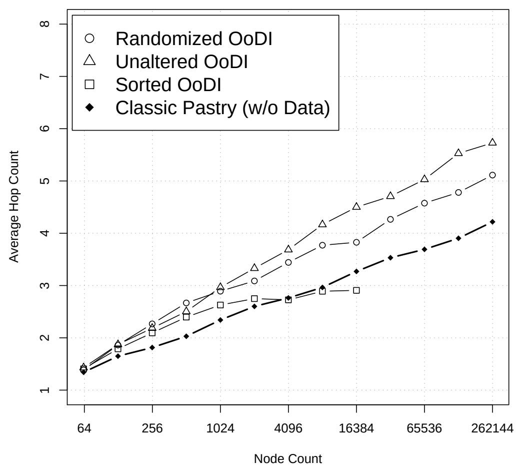
Since the load-balancing algorithm requires global knowledge of the least loaded node, which might not be available in a distributed P2P environment, we performed two different tests: One where we simply assumed global knowledge and one, where we used only local knowledge about the load of nodes contained in each node's routing table.

Figure 3.5 shows the resulting Lorenz curves of storage load distribution after data insertion for about 16k and 65k nodes. It depicts what fraction of nodes stores what fraction of data. The area between the bisecting line and the curve provides a scalar load imbalance measurement. The outstanding good performance of the Ganesan-Balancing itself is no surprise, since the algorithm provides a proven upper bound on the ratio of least to most loaded node. The interesting observation is, that the use of local knowledge indeed suffices to reach a load distribution similar to the one with global knowledge (confirming the prediction made in [7] that a sampling of $\mathcal{O}(\log n)$ random nodes should suffice). In contrast to the former, the load distribution of the ordinary DHT turns out to be poorly balanced and tends to get worse as the node count increases.

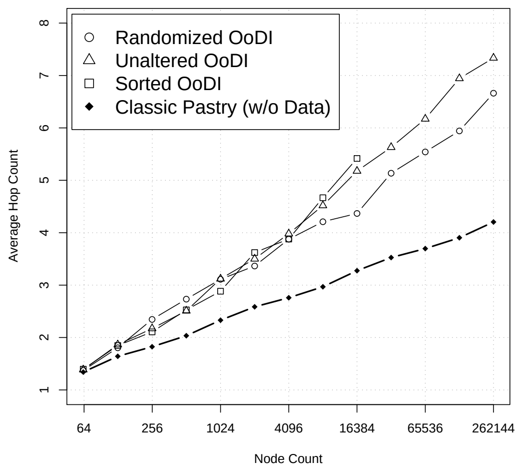
Routing Performance

After presenting the benefits, we have to discuss the impacts of the alphanumerical overlay on the underlying P2P network. As already discussed, Reorder operations cause the movement of nodes to a certain area of the numerical pastry ring resulting in a non uniform distribution of nodes. We examined the routing performance, meaning the number of hops needed to route a message from the source to its responsible node. Furthermore we discuss the impact of too many reorder operations on the routing table size as already mentioned in Section 3.4. The strength of the impact depends on the OoDI where sorted is a worst case and randomized is assumed as best case scenario.

In particular we have tested the average path lengths for the classic numerical routing and the new alphanumeric routing as well as the average memory consumption per node for different numbers of nodes and the three different OoDIs. The average numerical path length has been measured by sending $10,000$ test messages from randomly picked nodes towards the numeric ID of another randomly chosen node and averaging the resulting lengths afterwards. This simulates lookups to elements that hit the responsibility ranges of the nodes we route to. The average alphanumerical path length has been measured by sending $10,000$ test messages as well, but this time towards a randomly chosen name of a data element (that has been previously inserted).



(a) Numerical Routing



(b) Alphanumerical Routing

Figure 3.6: Routing Performance.

Figure 3.6a depicts the recorded average path lengths using numerical routing depending on the number of nodes in the network. It shows one curve for each OoDI tested and one curve with the tested numerical routing performance of a classic Pastry network (without data insertion and load balancing) for comparison purpose. The curve for the *sorted* OoDI ends with a node count of 16, 384 due to a scalability problem (which we'll discuss below) that prevented us from simulating higher node counts.

All of the three curves show an average path length that grows only sublinear with the number of nodes in the network. The *unaltered* and *randomized* OoDI curves show performances that are, however, worse than the one of a classic pastry, but with a maximum surcharge in path length of only $\approx 35\%$ in a network of 262, 144 nodes (compared to the classic pastry performance). The *sorted* OoDI curve shows a path length that tends to perform better than the classic pastry with increasing network size. This can be explained with a heavy increase in the average routing table population (see below). In general the different path lengths can be explained with a degeneration of the uniform distribution of nodes in the numeric ID space. The peculiarity of this degeneration depends on the respective OoDI.

Figure 3.6b shows the average path lengths of the alphanumeric routing as presented, depending on the network size. All pictured curves tend to grow only sublinear with the node count, with the surcharge in hop count being reasonably small compared to the classic numerical Pastry routing (which is depicted in Figure 3.6b for comparison purpose as well). In particular, with a network size of 262, 144 nodes the average alphanumeric routing path was still less than twice as long as the average numeric routing path in a classic pastry (more precisely we encountered a maximum average surcharge of $\approx 74\%$ at that specific network size).

Routing Table Population

The memory consumption has been measured in terms of routing table population, that is by counting the existing links in each node's routing table and averaging the sum. This is a valid measurement, since the routing table is the only entity of a node's state whose size may change dynamically - the leafset of a node always has a fixed size. As one can see in Figure 3.7, our alphanumeric ID space extension can have a crucial impact on the memory consumption when the Ganesan on-line load balancing algorithm gets applied (note that both axis are in logarithmic scale). The figure shows three different curves that depict the average number of existing links in each node's routing table depending on the OoDI used, as well as one curve with the average table population of a classic Pastry network.

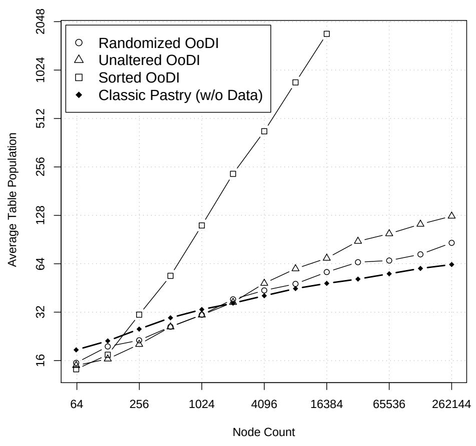


Figure 3.7: Routing-Table Population (Average Number of Entries)

The worst results were achieved by using the *sorted* OoDI, where a linear growth with the number of nodes can be observed. This happens due to a continuous insertion of data elements on a single node, which results in a continuous execution of the *Reorder()*-Operation that splits a narrow portion of the numerical ID space over and over. This leads to an elongation of the numeric IDs of the inserted nodes proportional to the number of data elements inserted.

Now those elongated IDs are ultimately responsible for a growth in the row count of the routing tables and thus also their population, limited only by the number of nodes in the network. This rapid growth also prevented us from simulating higher node counts than 16, 384 for the *sorted* OoDI, since the memory consumption of the entire network simulation thus grew quadratic in the number of nodes.

The average table populations of the two other OoDIs tested do however look promising. In particular, with a node count of 262, 144 nodes we observed an average population that was only roughly twice as high as the one of a classic Pastry in the case of the *unaltered* OoDI and even lower in case of the *randomized* OoDI.

3.6 Conclusion

In this chapter, we presented a simple way to use the routing structures of a ring based DHT network like Pastry to introduce an additional overlay using a total order of keys instead of hashing. The overlay enables load balancing as presented by Ganesan et al. Additionally, it is now possible to evaluate range queries, which is an important requirement of many application scenarios.

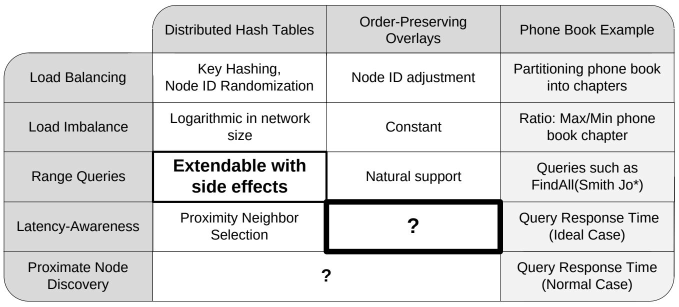
Our evaluation with simulations showed that the impacts of Reorder operations do affect routing performance, but not in a crucial way. Both, numerical and alphanumerical routing still reach their targets within only few hops. Problems arise if the insertion order of data is "bad", causing a lot of Reorders within a narrow area of the numerical ring. This results in an elongation of numerical ids and a clustering of nodes in this area. Choosing a randomized order of data insertion avoids this worst case.

4 The Papnet Overlay

Contents

PAPNET
proximity-aware peers

This chapter presents a new kind of overlay structure that does not rely on a DHT and can handle arbitrary skewed node and key distributions. In contrast to existing Overlays, the presented network *Papnet* provides range queries, allows for the implementation of Ganesan Online Load Balancing **and** is able to perform *Proximity Neighbor Selection* to optimize routing paths for latency (Figure 4.1) – a property not achieved by any previous hash-free overlay.



	Distributed Hash Tables	Order-Preserving Overlays	Phone Book Example
Load Balancing	Key Hashing, Node ID Randomization	Node ID adjustment	Partitioning phone book into chapters
Load Imbalance	Logarithmic in network size	Constant	Ratio: Max/Min phone book chapter
Range Queries	**Extendable with side effects**	Natural support	Queries such as FindAll(Smith Jo*)
Latency-Awareness	Proximity Neighbor Selection	**?**	Query Response Time (Ideal Case)
Proximate Node Discovery	?		Query Response Time (Normal Case)

Figure 4.1: Overlay Properties

4.1 Related Work

Many P2P overlay networks have been described in the literature with CAN [36], Chord [2] and Pastry [3] being some of the most prominent ones. The latter two both realize a discrete circular address space that is partitioned amongst the participating nodes. Efficient routing towards arbitrary addresses is provided by establishing links that skip exponentially growing portions of the address space. The discrete address space however puts major restrictions on applications that use the overlay: each participating node needs to have a static numeric id and each key that an application wants to store within the overlay has to be mapped onto such a discrete id as well.

Chord# [6, 67] is an extension to Chord which eliminates the need for a discrete address space and uniformly distributed addresses of nodes and keys (and thus the need for hashing). The authors show how efficient routing links can be constructed that skip exponentially growing numbers of nodes and thus allow for an arbitrarily skewed id space and range queries. The description however lacks detail in the definition of the id space and join algorithm, and often refers the reader to the original description of Chord. However, it must be pointed out that Papnet and Chord# are closely related in the way they set up links to skip exponential numbers of nodes.

Papnet allows for the implementation of the On-Line load balancing by Ganesan et al. [7] (GLB), which is a sophisticated load balancing algorithm that provides proven bounds on the global imbalance ratio of most- to least-loaded node. To our knowledge, Papnet is the first latency-aware hash-free overlay that natively allows for and demonstrates an implementation of the GLB.

SkipNet by Harvey et al. [5] is an overlay network that provides an alphanumeric address space as well and also allows for the implementation of the GLB. The major drawback of SkipNet is that participating nodes connect to each other in a strongly deterministic and symmetric fashion, leaving no space to prefer nearby nodes as routing links. In contrast to SkipNet, we will show how Papnet is able to reduce routing latencies by applying proximity routing, which is the preference of close by nodes as routing links. Such techniques have been studied and described in various publications and a good summary can be found in [27].

In its original form the GLB requires the knowledge of the globally least loaded node for the execution of the *Reorder()*-Operation. To find this node in a P2P overlay network is hard and expensive, if at all possible. Luckily [7] shows that the algorithm performs equally well if only the least-loaded node out of a sample set of $O(\log n)$ random nodes is chosen.

4.2 Papnet in a nutshell

Papnet is a P2P overlay that is able to construct an efficient routing structure independent of the actual key space. Looking at classic networks, we observe that all logarithmic node degree networks (such as Chord, Pastry, Kademlia, etc.) define two kinds of links:

- Local Neighbor-Links
  These links point to nodes that are direct neighbors in the address space and thus ensure the basic connectivity of the network. The more of these links a node establishes, the more failure resistant is the network.
- Far-Distance links
  These links point to nodes in increasing key space distances. They allow for efficient routing, since with each forwarding the distance to any arbitrary target key can be reduced by a constant fraction. Given that nodes are distributed uniformly in the key space, a typical routing path will have a length in $O(\log n)$.

Papnet also uses these two types of links, but constructs its Far-Distance links in exponentially increasing distances of *nodes* – not *keys*. This allows us to remove the assumption of uniform node and key distributions while preserving the $O(\log n)$ routing and memory complexities of classic networks.

In Papnet, we call the Far-Distance links the *Boundary Links*. They skip exponentially increasing numbers of nodes and are constructed strongly deterministically, which at first prevents latency-optimizations. However, Papnet introduces a third set of links called the actual *Routing Links*, which are located in-between any two succeeding boundary links. These links can be compared to Pastry's routing links, which are selected from certain ranges of the key space that contain exponentially increasing numbers of candidate nodes. In Papnet, the Boundary Links also define exponentially increasing ranges of candidate nodes, of which we can always choose the most proximate one to become a routing link. No matter which of the candidates are selected, the routing links still skip exponentially increasing numbers of nodes and therefore still allow for efficient, but now also latency-aware routing.

Figure 4.5 visualizes the links of a Papnet node. The long dashed arrows are the neighbor links that ensure stability and connectivity of the network. The outer short dashed arrows represent the constructed Boundary Links which skip 1, 2, 4, 8, ..., say exponentially increasing numbers of nodes. In-between any two succeeding boundary links you can see solid arrows that represent the actual Routing Links. Note how they still skip exponentially increasing numbers of nodes.

4.3 Formal Description

Papnet is an overlay network that provides a wrapping circular alphanumeric address space that can be imagined as a ring. In this section we will define this id space in detail and continue with a description of Papnet's topology and routing algorithm. For the sake of simplicity, we will at first describe a static network where no node arrivals or departures nor failures occur. The handling of arrivals, departures and network stabilization will be covered in the subsequent sections.

Key Space

The ID space of Papnet is an infinite linear Domain $\mathcal{K}$ (keys). Each **key** $k \in \mathcal{K}$ is a sequence of characters from a globally known **character set** $\mathcal{C}$. The **smallest character** in $\mathcal{C}$ is denoted by $\perp$, formally defined by the property $\perp \in \mathcal{C} : \forall c \in \mathcal{C} : \perp \leq c$. In particular, $\perp$ is a special zero character that may be arbitrary often appended to a key without changing the uniqueness of the key.

$$\mathcal{K} := \{c_1 \cdot c_2 \cdots c_i \mid c_{1..i} \in \mathcal{C}\}$$
$$\forall k \in \mathcal{K} : k = k \cdot \perp^+.$$

With the help of $\perp$ we can treat any two keys as if they had equal digit length. Now we can define a total order $\prec$ on $\mathcal{K}$ in the sense that $k_1 \prec k_2$, iff $|k_1| = |k_2|$ and k_1 is smaller than k_2 according to a classic lexicographical order. With this total order we are able to define wrapping **ranges** and **half-open ranges** of keys that use a directional parameter $d \in \{-1, 1\}$:

$$\forall k_1, k_2 \in \mathcal{K} : \mathbf{Keys}(k_1, k_2) := \begin{cases} \{k \in \mathcal{K} \mid k_1 \preceq k \preceq k_2\} & \text{if } k_1 \preceq k_2 \\ \{k \in \mathcal{K} \mid k_1 \preceq k \vee k \preceq k_2\} & \text{otherwise} \end{cases}$$

$$\forall d \in \{-1, 1\} : \mathbf{KeyRange}^d(k_1, k_2) := \begin{cases} \text{Keys}(k_1, k_2) \backslash \{k_2\} & \text{if } d = 1 \\ \text{Keys}(k_2, k_1) \backslash \{k_2\} & \text{otherwise} \end{cases}$$

Since $\forall k \in \mathcal{K} : k = k \cdot \perp^*$, we gain the useful property that for any two distinct keys $k_1, k_2 \in \mathcal{K}$ a **fresh key** can be constructed, which lies in-between them:

$$\forall k_1, k_2 \in \mathcal{K}, k_1 \neq k_2 : \exists k \in \text{Keys}(k_1, k_2) : k_1 \neq k \neq k_2$$

The above definitions describe the key space properties of Papnet formally. Figure 4.2 illustrates these properties for a concrete character and key set.

Character Set:	$\mathcal{C} = \{\perp, a, b, c, d\}$
Object Keys:	$\mathrm{O} := \{bad,\ cab,\ dad\} \subset \mathcal{K}$
Total Order:	$bad \prec cab = cab\perp = cab\perp\perp \prec dad$
Key Ranges:	$\mathrm{Keys}(dad,\ bad) \cap \mathrm{O} = \{bad,\ dad\}$
	$\mathrm{KeyRange}^{-1}(bad,\ cab) \cap \mathrm{O} = \{bad, dad\}$

Figure 4.2: Example Domain Properties

Node Space

The set of all existing nodes is denoted by $\mathcal{I}$. Each node $i \in \mathcal{I}$ possesses an arbitrary **key** $i.\mathrm{key} \in \mathcal{K}$ as well as a **unique identifier** $i.\mathrm{uid} \in \mathbb{N}$ that makes him distinguishable from any other node (and can be chosen at random). Having the unique identifier as a tiebreaker, we can define a **total order** $\prec$ on $\mathcal{I}$ as well:

$$\forall i, j \in \mathcal{I} : i \prec j := \begin{cases} i.\mathrm{key} \prec j.\mathrm{key} & \text{if } i.\mathrm{key} \neq j.\mathrm{key} \\ i.\mathrm{uid} < j.\mathrm{uid} & \text{otherwise} \end{cases}$$

Figure 4.3 shows an example for node names and unique identifiers. The nodes (circles) are depicted from left to the right with respect to the order $\prec$ on $\mathcal{I}$.

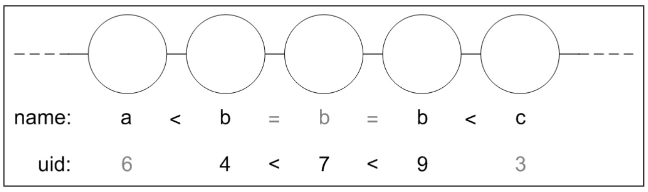


Figure 4.3: Example: Node identifiers and order

As for keys, we define ranges and half-open ranges of nodes:

$$\forall i, j \in \mathcal{I} : \mathbf{Nodes}(i, j) := \begin{cases} \{x \in \mathcal{I} \mid i \preceq x \preceq j\} & \text{if } i \preceq j \\ \{x \in \mathcal{I} \mid i \preceq x \vee x \preceq j\} & \text{otherwise} \end{cases}$$

$$\forall d \in \{-1, 1\} : \mathbf{NodeRange}^{d}(i, j) := \begin{cases} \mathrm{Nodes}(i, j) \backslash \{j\} & \text{if } d = 1 \\ \mathrm{Nodes}(j, i) \backslash \{j\} & \text{otherwise} \end{cases}$$

A **Neighbor** $N_k(i)$, $k \in \mathbb{Z}$ of a node i is the k-closest node to i with respect to $\prec$. It is thus the $|k|$'th node in the direction given by $\operatorname{sgn}(k)$. More precisely, we define $N_k(i)$ implicitly like this:

$$\forall i \in \mathcal{I}, k \in \mathbb{Z} : |\text{NodeRange}^{\operatorname{sgn}(k)}(i, N_k(i))| = |k|$$

Conventionally, we call $N_1(i)$ *clockwise* and $N_{-1}(i)$ *counter-clockwise* neighbor of i.

Neighbor-Links: Each node knows its immediate neighborhood, which is all k-closest nodes with $-\frac{l}{2} \leq k \leq \frac{l}{2}$ for a fixed constant $l \in \mathbb{N}$:

$$\forall i \in \mathcal{I}, -\frac{l}{2} \leq k \leq \frac{l}{2} : i.neighbor_k := \mathcal{N}_k(i)$$

These links ensure the basic connectivity of Papnet and guarantee routing termination.

Boundary-Links: Besides the neighbor-links, Papnet establishes sets of links of a second type which skip exponentially growing numbers of nodes and are used to detect the skewness of the id space. They are defined recursively in the following way:

$$\forall d \in \{-1, 1\}, k \in \mathbb{N} : i.boundary_k^d := \begin{cases} i.neighbor_d & \text{if } k = 0 \\ (i.boundary_{k-1}^d).boundary_{k-1}^d & \text{otherwise} \end{cases}$$

Each node possesses two sets of boundary-links. The directional parameter d controls which one of the neighbors is taken to serve as the first boundary-link; higher boundary-links are constructed from lower boundary-links residing on other nodes as can be seen in the above definition. Figure 4.4 shows example boundary-links.

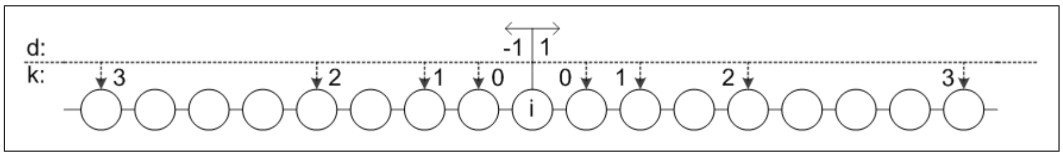


Figure 4.4: Boundary-Links for $d = 1$ and $d = -1$

Since $\mathcal{I}$ is a finite set of nodes there will eventually be a first link that wraps around in the circular node-space. The index of the directly preceding link is what we call the **maximum boundary index** $\kappa = \lceil \log_2 |\mathcal{I}| \rceil - 1$.

It is easy to observe, that the number of nodes that lie between two succeeding boundary links grows exponentially: $\forall i \in \mathcal{I}, d \in \{-1,1\}, k < \kappa$:

$$|\text{NodeRange}^d(i, i.boundary^d_k)| = 2^k$$
$$|\text{NodeRange}^d(i.boundary^d_k, i.boundary^d_{k+1})| = 2^k$$

Routing-Links: Each node further possesses sets of so called *Routing-Links*. These links can be chosen arbitrarily and are constrained only to the corresponding *boundary-links*. They must satisfy the following condition:

$$\forall i \in \mathcal{I}, d \in \{-1,1\}, k < \kappa : i.route^d_k \in \text{NodeRange}^d(i.boundary^d_k, boundary^d_{k+1})$$

Note that the number of nodes that each routing link skips still grows asymptotically exponentially:

$$\forall i \in \mathcal{I}, d \in \{-1,1\}, k < \kappa : |\text{NodeRange}^d(i, route^d_{k+1})| \geq 2^k$$

The freedom in node choice greatly allows for locality optimizations (e.g. by preferring nodes with low latency). How these nodes are found will be described in Section 4.4.

Figure 4.5 shows a complete example of a node's state in a Papnet with neighborhood size $l = 4$. The nodes the routing-links point to are highlighted as grey circles.

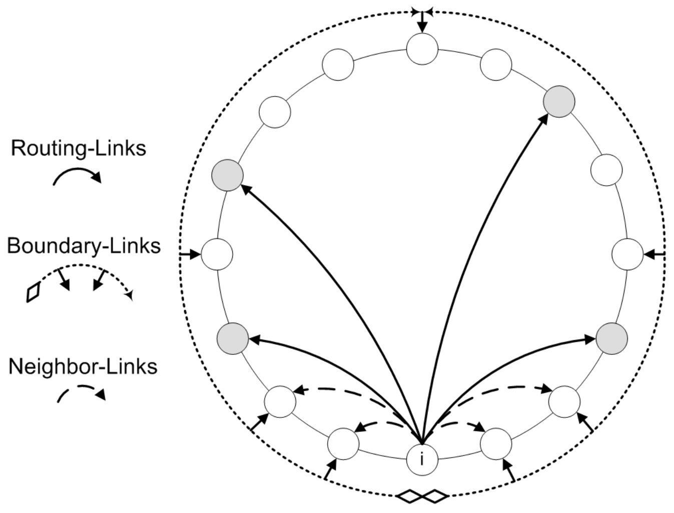


Figure 4.5: Example: State of a node in Papnet. ($l = 4$)

Responsibility and Routing

Each node in Papnet is responsible for the range of names that lie between the name of its immediate counter-clockwise neighbor (inclusive) and its own name (exclusive). As simple as this might sound, the precise definition is a bit more complicated due to the fact that Papnet allows multiple nodes to have the same name:

$$\forall i \in \mathcal{I}, j = i.\text{neighbor}_{-1} : \text{Resp}(i) := \begin{cases} \text{NameRange}^1(j.\text{name}, i.\text{name}) & \text{, if } j.\text{name} \neq i.\text{name} \\ \emptyset & \text{, else if } j.\text{uid} \leq i.\text{uid} \\ \mathcal{W} & \text{, else} \end{cases}$$

The special case $\text{Resp}(i) = \mathcal{W}$ only occurs, when all nodes have the same name and i is the node with the lowest uid.

Papnet employs a very simple and straightforward routing algorithm: A message is delivered locally, if the destination is within the local responsibility range or otherwise forwarded to the closest known node whose name succeeds the destination. The pseudocode is shown in Algorithm 4.

Algorithm 4: $i.\text{RouteMsg}(\text{dest} \in \mathcal{W}, \text{msg})$

```
1 $R := \{i\} \cup \bigcup i.\text{route}^*_* \cup \bigcup i.\text{neighbor}_*$
2 $Q := \{q \in R \mid \text{dest} \prec q.\text{name}\}$
3 if $Q = \emptyset$ then $Q := R$
4 $j := \min_{\prec}(Q)$
5 if $j \neq i$ then $j.\text{RouteMsg}(\text{dest}, \text{msg})$
6 else $i.\text{Deliver}(\text{msg})$
```

The usage of neighbor-links ensures that the message will eventually terminate at the node being responsible for the destination name. The routing-links on the other hand ensure that the routing path will have an efficient length, since the distance to the destination node is at least lowered by half with every hop in the average case. In the worst case it is only lowered by $\frac{1}{4}$ per hop, but will nevertheless stay logarithmic with less than $\log_{\frac{3}{4}} \frac{1}{n}$ hops in a network of n nodes.

4.4 Arrivals, Departures, Maintenance

This section covers the bootstrapping of new nodes and the maintenance- and repair-mechanism of Papnet, including pseudo-code algorithms based on the formal definitions of the previous chapter.

Uniqueness and Propagation of names

Papnet allows nodes to have the same name due to the dynamic nature of its address space: Names of nodes are not static and may change over time. Even if nodes would try to avoid names that already exist in the network, duplicated names may occur because network communication is unreliable and no node has a complete and correct view on the global network state. This is why we explicitly allow nodes to have the same name.

But this does not mean that equal names are desirable in Papnet. In fact it's the opposite: Each node in Papnet periodically checks, whether its name equals the name of its immediate clockwise neighbor. If that is the case, it tries to rename itself by choosing a fresh name out of the range between the name of its counter-clockwise neighbor and its own old name. This renaming mechanism ensures that all nodes will eventually possess unique names. Algorithm 5 shows the pseudocode for this periodic name correction.

Algorithm 5: i.CorrectName()

```
1 if $i.name = i.neighbor_1.name \neq i.neighbor_{-1}.name$ then
2     Let freshName $\in \mathrm{Resp}(i) \setminus \{i.\mathrm{neighbor}_{-1}.\mathrm{name}\}$
3     $i$.name = freshName
```

Since nodes are allowed to change their names at runtime, other nodes need to learn these new names. In Papnet, we use passive propagation where each keepalive-message (see the following sections) contains a hash-value of the sender's name. The recipient is able to compare this value with a hash-value computed from the sender's last known name. If the values differ, the recipient requests a name update from the sender.

Bootstrapping

An important part of Papnet is the bootstrapping algorithm. Since the system should be able to recover even in the presence of massive node failures, it must be ensured that any set of arbitrarily related failing nodes will be distributed uniformly along the ring of nodes. By placing a new node next to a node selected randomly from the set of existing nodes, Papnet ensures that failures of whole id space segments are very unlikely.

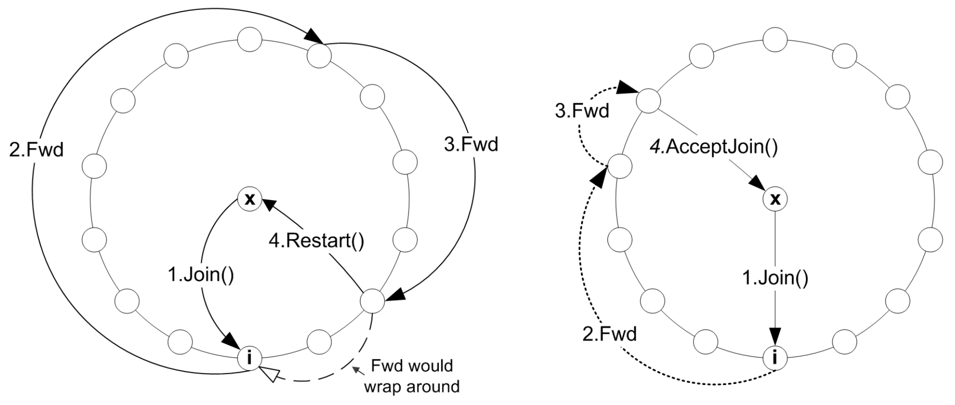
Classic overlays like Chord or Pastry place new nodes implicitly on (quasi-)random positions in node space by assigning them uniformly distributed ids in their finite id space. This method however does not work in Papnet, since its id space is infinite and can be arbitrarily skewed and thus provides no solid basis for the determination of a random node position. The boundary-links, however, can be utilized to reach a random position in node space.

A joining node x contacts an arbitrary first node i_{first} which is already a member of the network and sends him a join request. i_{first} then initiates a forwarding chain by a call to i_{first}.ForwardJoin($x, i_{\text{first}}, \kappa$) with κ being the index of its highest boundary link. Algorithm 6 shows the Pseudocode for *ForwardJoin*.

Algorithm 6: i.ForwardJoin(x, $i_{\text{first}} \in \mathcal{I}$, $k \in \mathbb{N} \cup \{-1\}$)

```
1  if $k \geq 0$ then
2      Let $i_{\text{next}} := i$
3      if $BinaryRandom() = 1$ then $i_{\text{next}} := i.\text{boundary}^1_k$
4      if $i \in Nodes(i_{first}, i_{next})$ then $i_{next}$.ForwardJoin($x, i_{\text{first}}$, $k-1$)
5      else $x$.RestartJoin()
6  else if $Resp(i) \neq \emptyset$ then
7      $x$.AcceptJoin($i$.name, ...)
8      Let freshName $\in$ Resp($i$)\{$i$.neighbor$_{-1}$.name}
9      $i$.name := freshName;
10 else $i$.neighbor$_{-1}$.ForwardJoin($x, i_{\text{first}}$, -1)
```

Any node i executing ForwardJoin(x, i_{first}, k) will forward the request to its k'th boundary-link with a chance of 50% (lines **1-3**). With each forwarding k gets decremented (line **4**) and once $k = -1$ the forwarding terminates. i will then choose a fresh name from its responsibility-range and let the new node join under i's former name (lines **7-9**). In the rare case that i is responsible for an empty range, it will forward the join message to its counter-clockwise neighbor (line **10**). Note that for every node j on the ring, a pattern of random forwardings (line **3**) exists that terminates at j. Furthermore, if we demand that the join-process will be canceled and restarted if a forwarding occurs that wraps around the ring (by crossing the start node i_{first}) (line **5**) we will reach every node on the ring with an equal probability.



(a) Failing (b) Succeeding

Figure 4.6: Join sequences. ($\kappa = 3$)

The probability p that a single join sequence fails can be bound to $p \leq 0.5$, since it depends on the probability that a forwarding occurs at stage $k = \kappa$. Otherwise, the sum of nodes that can be skipped by forwardings at the stages $k < \kappa$ is smaller than the total number of nodes in the ring and the join must therefore succeed. The probability that m consequent join sequences fail is thus bound by $p^m \leq 0.5^m$. Therefore, the repetitive join procedure will eventually (and quickly) succeed (the expected number of attempts is ≤ 2). Figure 4.6 shows examples for a failing (4.6a) and succeeding (4.6b) join sequence.

The last node i_{last} that receives and finally answers the join request sends its entire state to the new node. Since the new node is going to become a direct neighbor of i_{last} the sets of boundary- and neighbor-links it receives provide an excellent initial state.

Neighbor-link maintenance

To preserve a globally connected ring of nodes, each node periodically checks the liveness of its immediate neighborhood by sending keepalive-messages every δ_{neighbor} seconds to each of its neighbors $j = i.\text{neighbor}_k$ with $-\frac{l}{2} \leq k \leq \frac{l}{2}$. Along with this ping message, i passes k, indicating that i knows j as its k-closest neighbor. If in turn j knows i as its $(-k)$-closest neighbor, everything is fine and j will reply with a simple acknowledgement message. If on the other hand i is not the $(-k)$-closest neighbor of j (or even unknown to j), the reply will contain the set of all neighbors j knows of.

This simple mechanism ensures the detection of failed nodes in the neighborhood and that the arrival of new nodes will get propagated properly. Algorithms 7 and 8 show pseudocodes for the above mechanism. The command *VerifyLiveness()* used in the pseudo-code stands for a liveness-check by a simple ping message.

Algorithm 7: i.RecvNeighborPing($j \in \mathcal{I}, k \in \mathbb{Z}$)

```
1 $i$.InsertNeighbor($j$)
2 if $i.neighbor_{-k} = j$ then $j$.RecvNeighborPong($\emptyset$)
3 else $j$.RecvNeighborPong($\{i.\text{neighbor}_k \mid -\frac{l}{2} \leq k \leq \frac{l}{2}\}$)
```

Algorithm 8: i.RecvNeighborPong($M \subseteq \mathcal{I}$)

```
1 foreach $j \in M$ do if $\nexists k : i.neighbor_k = j$ then
2     if $j$.VerifyLiveness() then $i$.InsertNeighbor($j$)
```

Boundary-link maintenance

Boundary-Links are reconstructed periodically every $\delta_{boundary}$ seconds in an iterative fashion. The first boundary-link $i.\text{boundary}_0^d$ with $d \in \{-1, 1\}$ is always an immediate neighbor of i. The k'th boundary-link $i.\text{boundary}_k^d$ can be constructed by asking the node $i.\text{boundary}_{k-1}^d$ for this node's own $(k-1)$'th boundary link. This can be done until either a wrapping in the circular address space is encountered or the asked node is not able to return any link. Algorithms 9 and 10 show the pseudocodes for the boundary-construction.

Algorithm 9: i.RequestBnd$_k^d$($j \in \mathcal{I}$)

```
1 $j$.ReturnBnd$_k^d$($i, i.\text{boundary}_k^d$)
```

Algorithm 10: i.ReturnBnd$_k^d$($j_{\text{cur}} \in \mathcal{I}, j_{\text{next}} \in \mathcal{I} \cup \{\text{null}\}$)

```
1 if $\forall k' < k : i.boundary_{k'}^d \in NodeRange^d(i, j_{cur})$ then
2     $i.\text{boundary}_k^d := j_{\text{cur}}$
3     if $j_{next} \neq null$ then $j_{\text{next}}$.RequestBnd$_{k+1}^d(i)$
```

In the presence of message loss or node failure, a node i might not receive a reply for a request sent to a node j_{next}. In this case, i will send an overlay-lookup request towards j_{next}.name to find an adequate boundary-link and the recursive procedure will continue.

Routing-link maintenance

The routing-links can be chosen with a large degree of freedom, since they are only constraint to their corresponding set of boundary-links. An empty routing-link $i.\text{route}_k^d$ will always be initialized when the boundary-links get rebuilt and will be set to its corresponding boundary-link $i.\text{boundary}_k^d$.

The routing-links are also periodically checked for liveness. Every δ_{route} seconds each node increases a cycling counter m and sends a ping message to every $j = i.\text{route}_m^d$ with $d = \pm 1$. When a routing-link has been detected to have failed, it is set to *null* and will be reinitialized once the next boundary-link rebuilding is performed.

There is room for optimizations here to fix failed routing-links faster, e.g. by caching addresses of suitable nodes, but we won't discuss them here. Our results show that fixing at the time when boundary-links are rebuilt suffices to keep a good routing performance in times of churn.

4.5 Latency Optimizations

One of the key features of Papnet is its proximity-awareness, which we understand as the preference of nearby nodes according to a "closeness"-function $distance : \mathcal{I} \times \mathcal{I} \to \mathbb{R}$. This function can be based on a-priori knowledge (e.g. geographic location) or be provided by an external source (e.g. ISP), but in the in the most common case, the values of this function are determined at runtime, being simply the round-trip-times between any two nodes. In this section we will describe the optimization of routing links based on the latter method, by using ping messages to determine the distance.

Finding close nodes

Each node i periodically tries to improve one of its routing links $j = i.\text{route}_k^d$. The set of valid candidate nodes that are suitable to replace j is defined by the following set:

$$R := \text{NodeRange}^d(i.\text{boundary}_k^d, i.\text{boundary}_{k+1}^d)$$

To find suitable close nodes in R we make use of the assumption that the routing links of each node are already close to that node.

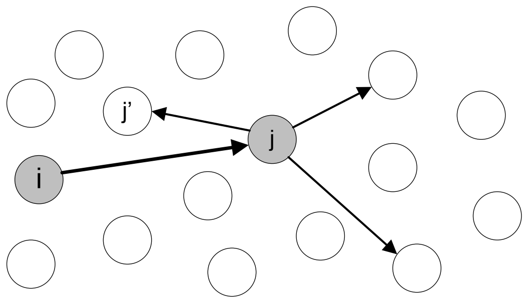


Figure 4.7: 2D geospatial view of the network

Figure 4.7 illustrates this assumption and depicts a geospatial view of the nodes. It shows the node i, its current routing link j and the routing links of j. In particular, one of these links (j') is also close to i and may replace j as a routing link of i.

Following the assumption of proximity transitivity, the node i will ask its current routing link j for *one* of its own routing links. Along with the request i passes the boundary index k. j then picks a random node s out of the set

$$Q := \{j.route_{k'}^{d'} \mid d' \in \{-1, 1\}, 0 \leq k' < k\}$$

and sends s's physical address to i. Figure 4.8 visualizes the nodes i and j and their boundary links, as well as the sets R and Q. Note that at half the nodes in Q (light grey nodes) are also in R.

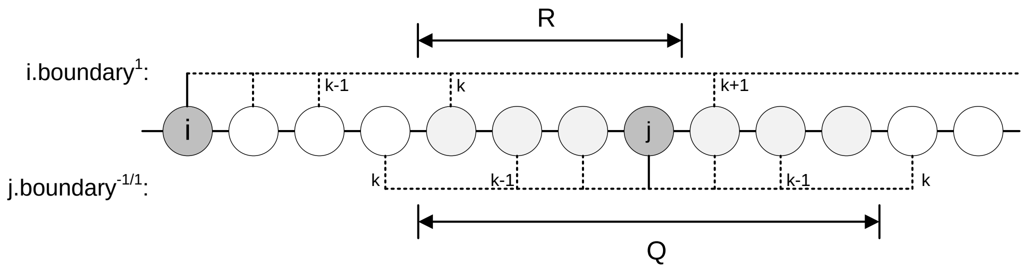


Figure 4.8: Logical view of the network

The advantage of this probabilistic discovery approach is that it does not require any transfer of node keys defining the candidate range. Such keys could easily dominate the message payload size, since Papnet allows for keys of arbitrary length. Instead, the presented approach requires only messages of constant size and is nevertheless very likely to return a suitable candidate.

After having received the address of a candidate j' Node i will simply send a ping message towards j' to measure its distance. Only if the distance is lower than that of the previous routing link j, it will ask for j's name ID and test if it indeed lies within the candidate range R. The overhead of the presented algorithm is extremely low since discovery comes at almost no costs, but is yet very effective, as the results in the next chapter will demonstrate.

Discovery Efficiency

We now need to show that the above mechanism will provide *good* candidates to i. In particular, we will show the probability that the returned link s lies within the candidate range R is at least 50%, since at least half the nodes in Q also lie within R.

This can be shown by looking at unique node pairs in Q:

$$\text{Let } m \in [0..k-1],\ m' = k-1-m \text{ so that } p = j.\text{route}_m^{-1} \text{ and } q = j.\text{route}_{m'}^{1}$$

Each such p and q lie in opposite directions to j on the ring and the union of all p and q is Q. We want to show, that at least one of them also lies in R. Therefore, we consider the range

$$S = \text{Nodes}(p, q) = S_p \cup S_q \text{ with } S_p = \text{Nodes}(p, j) \text{ and } S_q = \text{Nodes}(j, q)$$

We observe that $|S| = |S_p| + |S_q| - 1$ because both S_p and S_q contain j. Further we know that $|R| = 2^k$, $|S_p| \leq 2^{m+1}$ and $|S_q| \leq 2^{m'+1}$. Because both S_p and S_q contain j and the bounding nodes of S are p and q, the only way that neither p nor q are in R is that R lies entirely in S.

This in turn requires that $|S| \geq |R| + 2 = 2^k + 2$, which we will now show to be impossible:

(I) If $max(m, m') < k-1$ then $|S_p| + |S_q| \leq 2 \cdot 2^{k-1} = 2^k$.
(II) If $max(m, m') = k-1$ then $min(m, m') = 0$ and $|S_p| + |S_q| \leq 2^k + 2^1 = 2^k + 2$.

In both cases $|S_p| + |S_q| \leq 2^k + 2$ and thus $|S| \leq 2^k + 1$. This violates the requirement $|S| \geq 2^k + 2$ and thus R cannot lie entirely in S and must contain at least p or q.

Summing up, this means that at least half the nodes in Q are also in R. Therefore, the discovery mechanism will provide suitable candidate nodes with a reasonable probability of at least 50%.

4.6 Evaluation

To evaluate Papnet's performance we deployed a large-scale network on the students' computer lab of the math faculty of TU Berlin. We made use of 100 real machines and ran 500 virtual nodes per real machine, resulting in a total network size of 50,000 nodes. For communication we used UDP datagrams. The virtual nodes located on the same physical machine used a shared network layer to dispatch in- and outbound messages, but still communicated with each other using the network layer of the operating system.

In our tests we instantiated the adjustable parameters of Papnet in the following way: The neighborhood-size was set to $l = 16$ nodes. The neighbors were tested for liveness every $\delta_{\text{neighbors}} = 24$ seconds. The boundary-links were rebuild every $\delta_{\text{boundary}} = 60$ seconds. The routing links were incrementally tested for liveness and asked for replacement candidates every $\delta_{\text{routing}} = 5$ seconds. As character set $\mathcal{C}$ we simply used bytes s.t. $\mathcal{C} = [0..255]$. The first node of the network was given a random name consisting of 160 characters. The resulting average name length of all 50,000 nodes was thus also roughly 160 characters.

We evaluated Papnet by a test comprising 7 phases:

1. **Join-phase:** 50,000 nodes joined over 60 minutes ($833.\overline{3}$ joins per minute).
2. **Stabilization-phase:** No arrivals/departures occurred.
3. **Churn-phase:** 30,000 nodes joined and left (mixed) within 30 minutes (1000/min).
4. **Reorder-phase:** Random nodes sent reorder requests to a random boundary-link. 30,000 reorders were requested within 30 minutes (request rate = 1000/min).
5. **Rename-phase:** Random nodes changed their name within the bounds of their neighboring node's names. 30,000 renames were performed within 30 minutes (rename rate = 1000/min).
6. **50%-failure-phase:** Half the nodes failed at once.
7. **Shutdown-phase:** All nodes left (rate = $416.\overline{6}$/min).

Measurements of key system properties throughout these phases are depicted in Figures 4.9-4.11.

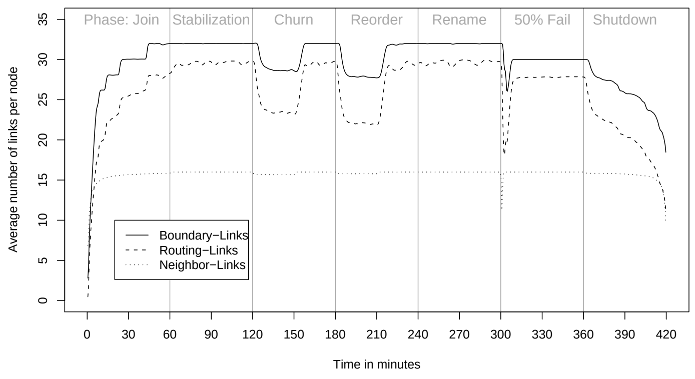


Figure 4.9: Average node state

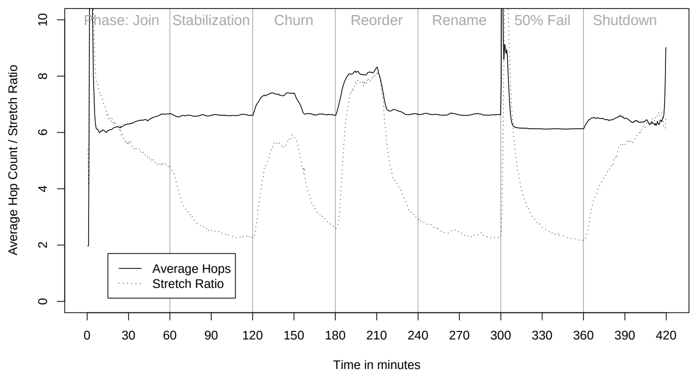


Figure 4.10: Average routing performance

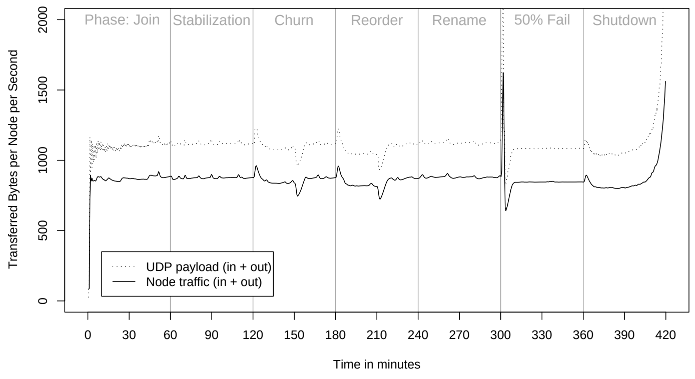


Figure 4.11: Average maintenance costs

Figure 4.9 shows the average node state in terms of link counts. Each node maintains only an efficiently small state throughout all phases. Even in the presence of massive node failure (phase 6), the system is able to recover very quickly.

We tested the routing performance after the actual test, based on the states that each node wrote to a logfile every minute. We simulated the routing of about 100,000 messages starting at a random node towards the name of another random node and recorded the average path length. For every forwarding to a nonexistent node we added one penalty hop. Figure 4.10 shows the encountered numbers of hops. As we can see, the average routing path length stays efficiently small throughout all phases and rises only slightly in times of churn.

To test the proximity-optimization, we could not utilize real latencies, since they were simply too small (less than 1ms). Instead, we assigned each node a random finite position $(x, y) \in [0..65535]^2$ and simulated latencies by calculating the Euclidean distance between two node's positions. Figure 4.10 shows the encountered stretch-ratios, which are calculated by dividing the sum of latencies of the routing path by the direct latency between first and last node. As we can see (especially in phase 2), the ratio quickly drops towards a value close to 2, which is optimal according to [27].

While performing very well in general, Papnet requires only little maintenance overhead. Figure 4.11 shows the average traffic (in- plus outbound) of a node we encountered throughout the phases. As you can see, each node's traffic (solid line) stays below 1kB most of the time. Since the virtual nodes on each real machine used a shared message-dispatching layer (which also handled packet fragmentation) the figure shows two graphs: The traffic encountered at node-level (solid line) and the actual UDP payload (dashed line), which is a little higher due to virtual node addressing and fragmentation overhead.

Ganesan Load Balancing

On top of Papnet we implemented a simplified version of the GLB, which only considers the clockwise instead of both neighboring nodes in the *NeighborAdjust()*-operation, because a shift of load to the clockwise neighbor involves only a local change of the own name. A shift to the counter-clockwise neighbor would require a message being sent to that neighbor, asking it to change its name. The initiating node would have to wait for a reply and would therefore need to save state information. This can be avoided through our restriction to consider only the clockwise neighbor. The algorithm itself is only slightly (if at all) affected by this change, as can be seen in the test results.

The GLB *Reorder()*-operation requires a random node sample set of size $O(\log n)$. To find such a set, we first lookup a random node in the ring, using the very same forwarding strategy as in the bootstrapping algorithm (see 4.4). Then we use this node's boundary-links to sample the least loaded node from, because they provide the right number of nodes and a good distribution: No two node's sets of boundary-links are equal and since nodes are placed at random positions in node space, they provide a truly random sample set.

As a scenario we have a distributed Resource Description Framework (RDF) database like BabelPeers [58] in mind. RDF information is given through triples with subject, predicate, and object (SPO). The used data set is an artificial RDF-graph generated by the Lehigh University Benchmark[65] (Lumb-10) comprising 1,272,764 triples. Each triple is one data unit and must be accessible if only one part of a triple is known. Therefore, it is inserted under 3 different names: (SPO), (POS), (OSP). Thus, information is also accessible using range queries with wild cards for the unknown components if only one or two components are known.

Figure 4.12 shows the final load distribution after inserting the 3,818,292 data objects in a network of 50,000 nodes. The old method BabelPeers and other DHT-based RDF-stores use to store RDF information is to hash each triple to 3 different keys calculated from S, P and O. For comparison purpose we also inserted the data into a simulated Pastry network consisting of 50,000 nodes using this old method and show the resulting (bad) load distribution in Figure 4.12 as well.

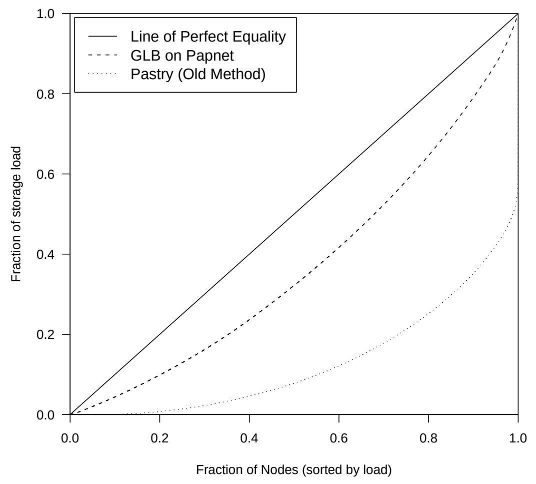


Figure 4.12: Lorenz curve of storage load

4.7 Conclusion

In this chapter we presented a detailed description of Papnet, the first proximity-aware order-preserving overlay network allowing for the implementation of Ganesan on-line load balancing. We presented a unique join algorithm which ensures a random node distribution and provides Papnet with high fault-tolerance properties. We demonstrated that Papnet is able to scale up to 50.000 nodes (and beyond) while producing maintenance costs of only about 1 kilobyte per second per node. Further, we presented a low-cost proximity-optimization strategy being solely applicable to Papnet, which our results show to perform nearly optimal and enables to reach every participating node with a latency being only roughly twice the direct latency.

5 Discovery of Proximate Peers

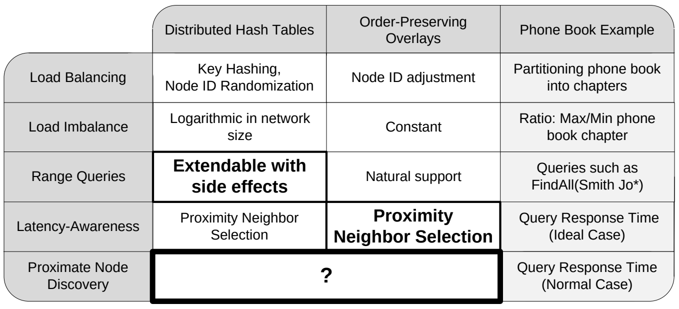
Contents

5.1 Introduction

In this chapter, we will show how Papnet has been extended to provide a fast discovery of most proximate nodes and to guarantee eventual optimality of its routing tables (Fig. 5.1). In particular, we make the following contributions:

- We show that hash-free overlay networks are able to provide efficient path latencies as good as hash-based overlays (using Pastry as a representative).
- We introduce an efficient algorithm to discover proximate peers.

We compare Papnet and Pastry using simulations based on generated as well as real latency data measured on the Internet.



	Distributed Hash Tables	Order-Preserving Overlays	Phone Book Example
Load Balancing	Key Hashing, Node ID Randomization	Node ID adjustment	Partitioning phone book into chapters
Load Imbalance	Logarithmic in network size	Constant	Ratio: Max/Min phone book chapter
Range Queries	**Extendable with side effects**	Natural support	Queries such as FindAll(Smith Jo*)
Latency-Awareness	Proximity Neighbor Selection	**Proximity Neighbor Selection**	Query Response Time (Ideal Case)
Proximate Node Discovery	**?**		Query Response Time (Normal Case)

Figure 5.1: Overlay Properties

5.2 Related Work

A crucial property of P2P overlays is their awareness of the underlay network. Without routing tables being optimized for proximity, the total routing path latency is likely to grow linearly with the path length (number of hops).

A common strategy to optimize path latencies in overlay networks is called *Proximity Neighbor Selection* (PNS) [55]. The key idea is to put loose constraints on the routing links, so that multiple nodes qualify for a routing entry and the most proximate one can be chosen. This technique has been intensively studied by Castro et al. in [68] and utilized in the locality-aware overlay network Pastry [3]. However, Pastry makes assumptions about the proximity of nodes at join time and optimizes its routing links only when there is message traffic. It is thus not able to guarantee eventual optimality (see section 5.3) of its routing tables.

In [69] Milic and Braun describe an algorithm for topology aware choice of peers in overlay networks. Their approach uses a gravity analogy to construct a local network view called the *fisheye view* of constant size. The fisheye view contains many near and few far away nodes and the authors argue that this property makes a fisheye view the ideal view that an end-system should have in any overlay network. In that sense, the routing tables of Pastry as well as Papnet do represent fisheye views. However, their paper unfortunately does not evaluate the scalability of the approach, which would be of special interest since their views are of constant size. Furthermore, it lacks a discussion on how to embed an ID space into their overlay network that allows for the implementation of efficient DHT-like functionality.

In [70] Wu et al. propose to improve locality by proximity-aware assignment of IDs. They use a peers Autonomous Systems Number (ASN) to generate the higher bits of its ID, resulting in a clustering of nodes from the same autonomous system (AS) in the ID space. While this method indeed leads to shorter path latencies, the authors do not discuss the impact of the clustering on the network's fault tolerance properties. Failing nodes from the same AS are no longer distributed uniformly in the ID space, but most likely clustered in near regions, that way failures of complete ID space partitions are more likely to occur.

There are only few overlay networks that explicitly avoid hashing. Skipnet by Harvey et al. [5] is an example of such a hash-free overlay, but it does not support proximity optimizations at all. The closest relative of Papnet is Chord# by Schütt et al. [6], a hash-free overlay that sets up links in exponentially growing node distances similar to Papnet, but is not optimized for node proximity as well. Within the class of hash-free overlay networks, Papnet is the first that is able to perform PNS.

5.3 Papnet

In chapter 4, we introduced a novel overlay network called *Papnet.* In contrast to classic DHTs, Papnet supports the implementation of a DHT-like storage functionality without the need of hashing tuple names onto address keys. In fact, Papnet decouples the address space from the logical node space and allows for an arbitrarily skewed address space. This becomes possible, because Papnet constructs its efficient routing links explicitly and independently from the actual keyspace, while the efficiency of routing links in networks like Chord or Pastry depends on the assumption that all nodes are distributed uniformly in key space. Our approach has several advantages: It allows for efficient range queries, as well as the relocation and renaming of nodes. Papnet therefore is ideally suited for the implementation of *Ganesan Online Load Balancing* [7], which provides a *constant* global load imbalance ratio.

Structure

A *Papnet* is an overlay network consisting of n participants (nodes). Each node is free to choose an arbitrary address from a global address space (key space) $\mathcal{W}$. There exists a total order $<$ on $\mathcal{W}$ which allows to arrange all nodes in a ring shape and determine each node's successor and predecessor. Each node in Papnet is responsible for a unique partition of the key space $\mathcal{W}$ that is determined by the node's address i and the address of its predecessor j: $resp(i) = [j, i)$. The union of all partitions covers the complete key space. In contrast to other networks, the key space of Papnet is *infinite* and allows for keys of arbitrary length. Further, it has the unique property that between any two distinct keys a and b with $a < b$, a third key c can be constructed so that $a < c < b$.

Each Papnet node stores and maintains only an efficiently small set of links to other nodes of size $\mathrm{O}(\log n)$. Nevertheless, each node is able to send messages towards arbitrary addresses and the path taken by each message is guaranteed to consist of only an efficient small number of forwarding steps (hops), which is also in $\mathrm{O}(\log n)$. Further, Papnet uses a technique called *Proximity Neighbor Selection* [55] that allows for optimizing the latency induced by the forwarding steps.

The routing table of a Papnet node has three components: so-called *neighbor-links* which ensures the basic connectivity of the network and *boundary-links* that enable efficient routing. The third component are the proximity-optimized *routing-links.*

The *neighbor-links* of a node i point to the closest nodes preceding and succeeding i in the key space. These links are periodically tested for liveness and get properly adjusted in case of node arrival, departure or failure. The actual repair mechanism is trivial, so we refer the interested reader to a detailed description in [71].
A simple explicit scheme is used to construct so-called *boundary-links* in exponentially growing node distances. The boundary link at level 0 $B(i)_0$ of node i is defined to be its immediate neighbor. The boundary link at level 1 $B(i)_1$ is constructed by asking the node $j = B(i)_0$ for its own level-0 boundary link $B(j)_0$. The boundary link at level 2 $B(i)_2$ is constructed by asking the node $j = B(i)_1$ for its own level-1 boundary link $B(j)_1$. This procedure continues until a link is constructed that skips more than the total of nodes in the network. Thus, the number of boundary links is in $\mathrm{O}(\log n)$.
Using the *boundary-links*, efficient routing in terms of forwarding steps can be realized. However, the routing will not be efficient in terms of latency. This is because each boundary link $B(i)_x$ is constructed deterministically and the latency between i and $B(i)_x$ is expected to be the median latency δ between two arbitrary nodes in the network. Using only the boundary links for routing will therefore induce a total latency of $|p| * \delta$, where $|p|$ is the length of the path. To avoid these expensive routing costs, *Papnet* defines a third set of links called the (actual) *routing-links*. These links are very similar to the boundary links, except that each routing link is free to be any node in-between two succeeding boundary links.

Figure 5.2 illustrates the state of a Papnet node. It shows two views of the very same network: the key space view, where nodes are positioned on a uniform (unskewed) ring and the logical node-space view, where nodes are positioned in uniform distances on a skewed key space ring.

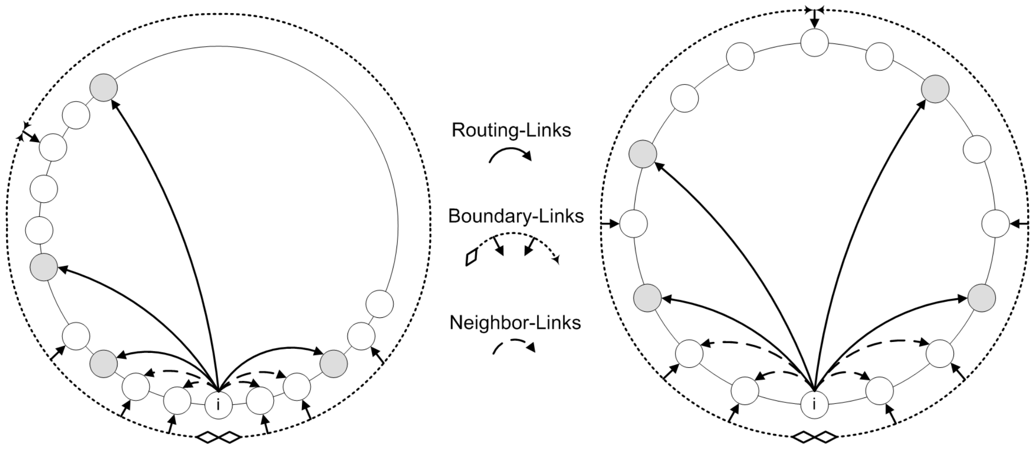


Figure 5.2: Key- and node-space view of a Papnet network.

5.4 Papnet Extensions

In chapter 4 we described a simple routing algorithm for Papnet that provides an efficient average path length of $\log_{\frac{4}{3}} n$ hops:

Any node i that receives a message for destination key d forwards the message to the node j taken out of i's neighbor- and routing-links that is the closest node succeeding d.

While this simple algorithm works very well, it does not take full advantage of Papnet's bi-directional routing links. Therefore, we will now refine the routing algorithm:

Any node i that receives a message for destination key d first checks if i or any of i's immediate neighbors is responsible for d. If that is the case, i delivers the message to the respective node. If not, i chooses the set of boundary links $B(i)^{left}$ or $B(i)^{right}$, so that $d \in [B(i)_k^{dir}, B(i)_{k+1}^{dir})$ and k gets minimal and forwards the message to $R(i)_k^{dir}$ or any known closer node that lies in-between the chosen boundaries.

The major difference to the previous routing algorithm is that the node x that is chosen for forwarding does not necessarily have to *succeed* the destination d anymore. Instead, it may be any node that lies inbetween the interval of boundary links $B(i)_k$ and $B(i)_{k+1}$ for the direction where the remaining distance is smallest. Algorithm 11 describes the new routing in a more precise way:

```
Algorithm 11: $i$.RouteMsg(d $\in \mathcal{W}$, msg)
1 if $d \in$ resp(i) then $i$.Deliver(d, msg)
2 if $d \in$ resp(nbr(i)) then nbr($i$).RouteMsg(d, msg)
3 Let dir $\in$ {left, right}, so that $d \in [B(i)_k^{\text{dir}}, B(i)_{k+1}^{\text{dir}})$ and $k$ is minimal
4 Let $C := \{x \in R(i) \cup N(i) \mid \text{x} \in [B(i)_k^{\text{dir}}, B(i)_{k+1}^{\text{dir}})\}$
5 Pick $j$ from $C$ where distance d($i$, $j$) is minimal
6 $j$.RouteMsg(d, msg)
```

The above algorithm guarantees termination after a maximum of $\lfloor \log_2 \frac{n}{2} \rfloor$ hops. This can be shown in the following way: Consider the first routing step on some node i: the maximum node distance s to the node being responsible for the destination key is $s = \lfloor \frac{n}{2} \rfloor$. The destination key therefore lies between two boundary links $B(i)_k$ and $B(i)_{k+1}$ where $k \leq \lfloor \log_2 s \rfloor$.

By forwarding the message to some node j that is located in-between these boundaries, the remaining node distance s' is lowered to be strictly smaller than 2^k. Therefore, in the next routing step j will consider an interval of boundary links $[B(j)_{k'}, B(j)_{k'+1})$ where $k' \leq \lfloor \log_2 s' \rfloor < \lfloor \log_2 2^k \rfloor = k$, pick one node out of that interval and forward the message.

The key observation is that with each routing step, the level k' that is used in the next step is strictly smaller than the currently used level k. We can therefore bound the maximum path length by $\lfloor \log_2 \frac{n}{2} \rfloor$.

Optimal proximity

The goal of Papnet's proximity optimization is to find the most proximate routing links. Any routing link of a Papnet node has to satisfy the condition of lying inbetween two of this node's boundary links. Thus, there exists a fixed deterministic set of candidate nodes for any routing link of a node. We say that a routing link of a node i is *optimal*, if there is no other candidate node that is closer to i according to a distance function $d(x, y)$.

All routing links are initialized with their corresponding boundary link: $R(i)_k := B(i)_k$ for all levels k, so that the structural condition $R(i)_k \in [B(i)_k, B(i)_{k+1})$ is satisfied. As nodes will learn about more proximate nodes from the set of candidates, they will replace their routing links with those nodes. However, this also means that the distance of any particular routing link can only decrease, but never increase. Thus, once a node has learned about all candidate nodes for one of its routing links, this link will be *optimal.* Once all routing links of a node are *optimal*, we say that the node is *optimal.* Finally the network is *optimal*, if all nodes are *optimal.*

We can guarantee the eventual *optimality* of the network, if we can ensure that all nodes will eventually learn about all candidates of any particular routing link. Eventual *optimality* can be achieved easily by successively testing all other nodes in the network. However, this trivial solution will not converge fast towards an optimal state. In contrast, Papnet uses a candidate discovery protocol that allows for fast optimality convergence and eventual optimality. Note that we only consider static networks that are not subject to churn (node arrivals and departures). However, even in dynamic networks convergence towards optimal tables can be guaranteed once no more churn occurs.

Optimization Protocol

Papnet utilizes the simple assumption that the routing-links $R(i)$ of any node i are close to i. Now if j is any routing link of i, then we expect that the routing links $R(j)$ are not only close to j, but also to i, since j is close to i. Therefore, whenever a node wants to optimize a routing link $j = R(i)_k$ on some level k, it will ask j for some node x that may be suitable to replace j in i's routing table.

The node j will return a random one of its routing links $x = R(j)_{k'}$, where $k' < k$. By restricting the level k', we ensure that the probability that x satisfies the structural requirement $x \in [B(i)_k, B(i)_{k+1})$ is at least 0.5, as we have shown in section 4.5.

Node i will therefore learn about a close node x that is likely to be from the set of candidate nodes for its routing link $R(i)_k$.

While this optimization technique already provides a fast discovery of nearby candidate nodes, it does not guarantee eventual *optimality* of the link. To guarantee that all possible candidates will eventually be discovered, we modified the above protocol slightly.

The node j that is asked by i now will immediately return the node x as described above only with a probability of 0.5. Alternatively, it will forward the discovery request to a random routing link $j' = R(j)_f$, where $f<k$. In that case, j' will proceed in the same way as j, either reply to i with a random $x = R(j')_{k'}$ where $k'<k$ or forward to some random $j'' = R(j')_{f'}$ with $f'<f$. In order to guarantee termination, we restrict the level f to get strictly smaller with each forwarding.

For every node in the candidate set, it is possible to construct a sequence of forwarding steps using only routing links of strong monotone decreasing level that will result in that very node to be discovered. The construction is similar to the routing algorithm: Consider the node j as above and some node x that we want to be discovered. Now since j and x are from the candidate set of the routing-link $j = R(i)_k$, we know that the node distance between both is $< 2^k$, because the candidate set is of size 2^k. Now there are two cases: either the distance is already also lower that 2^{k-1}, or j has a routing link $R(j)_{k-1}$ that points to a node whose distance to x is lower than 2^{k-1}. In the second case, $R(j)_{k-1}$ will be the first forwarding step on our path towards x, otherwise we stay on j. Now we consider the next lower distance 2^{k-2} and proceed in an analogue way, either forward or stay. Eventually, we will reach the destination node x by that strategy and the path will consist of forwardings using strongly decreasing levels of routing links. The randomization of forwarding finally ensures, that this path will eventually be taken.

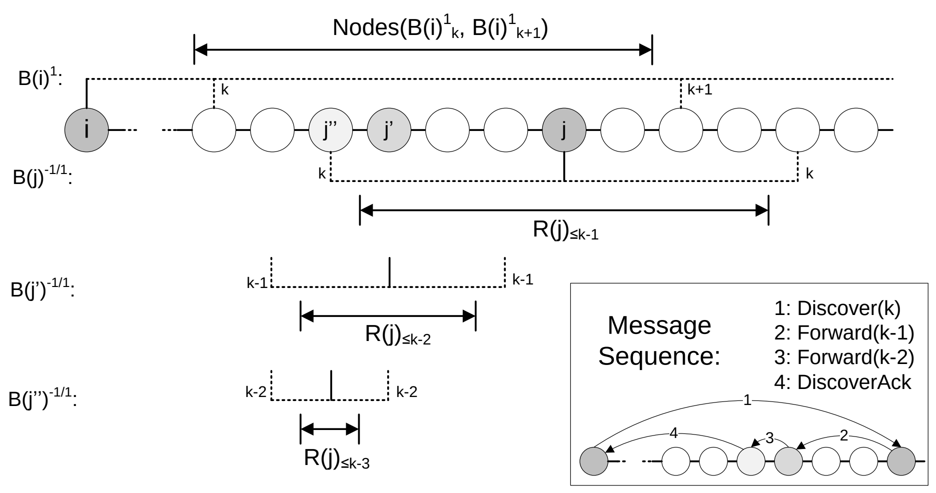


Figure 5.3: Optimization Example

Figure 5.3 illustrates the discovery procedure: Node i wants to discover a new routing link that lies in-between the boundaries $[B(i)^1_k, B(i)^1_{k+1})$. Thus, it sends a discovery message to its current routing link j, which in turn forwards it to a random own routing link j'. j' forwards the message even further to one of its own routing links j''. The strictly monotone-decreasing boundary index k ensures an eventual termination of forwardings. Along the path, the decision to forward a message is randomized and happens with a probability of 50%. In the remaining 50% of cases, a node will not forward, but answer directly to i.

The described optimization protocol ensures that most close nodes are found fast and that all candidates of any routing link will be discovered eventually. Therefore, the network must eventually converge to an *optimal* state.

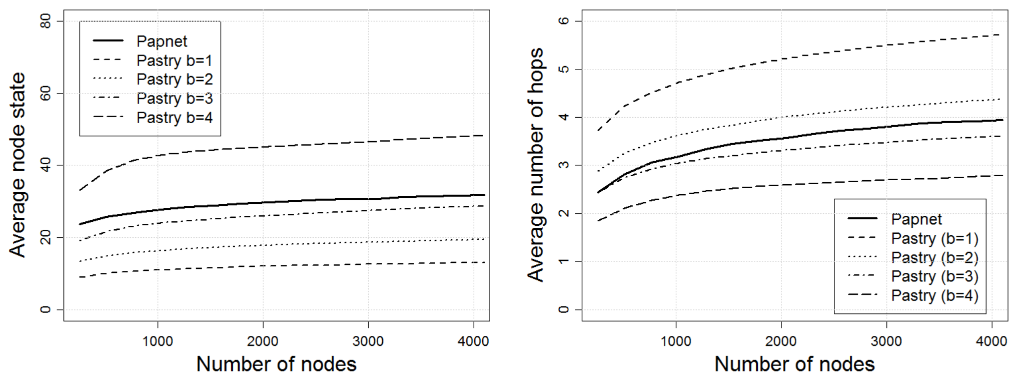
Costs

Each optimization step is expected to cause three messages. The first one is caused by the initial discovery request of i to one of its routing links j. The second message follows from the expected number of forwarding steps, which equals 1. This is because the probability that the forwarding terminates after 0,1,2... forwarding steps is $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}$... and thus the expected value calculates to $\sum_{i=0}^{\infty} \frac{i}{2^{i+1}} = 1.0$. The third message is the final reply containing the potential candidate x. In case latency is used to determine the distance $d(i,x)$, two more messages are required to measure the round-trip-time.

5.5 Evaluation

In this section, we evaluate the performance of Papnet and compare it with the well-known hash-based and proximity-aware overlay network Pastry. Both networks have been implemented as simulations in Java. Comparing Papnet and Pastry is not easy since both use different rules to setup their routing links. Pastry also has a configurable parameter b that has a large influence on the size of the routing table as well as the number of neighbor-links. We decided to instantiate Pastry with values of $b = 1 \ldots 4$ and 2^b neighbors each. In Papnet we used a fixed number of 16 neighbors. In all comparison tests, we filled each nodes routing table with optimal links using the following strategy: For any possible entry in the routing table, lookup the subset of nodes in the network that are possible candidates for that very entry and pick the most proximate one according to the latency metric. This leads to *optimal* routing tables in both, Papnet and Pastry.

In order to make both networks comparable, we measured the number of *unique* node links in each nodes routing table (which is the union of routing- and neighbor-links), the so-called *node state*. Figure 5.4 a) shows the encountered average node states for Papnet and four different instances of Pastry with network sizes ranging from 250 to 4000 nodes. We observe that Papnet has a slightly larger average node state than a Pastry network with $b = 3$, but fewer entries than one with $b = 4$. Note that Papnets *Boundary links* are not included in the node state, since these are not used for routing. Figure 5.4 b) depicts the average number of routing steps needed to route a message between arbitrary nodes. Clearly, paths in Papnet consist of more hops than in a Pastry network with $b \geq 3$. However, both figures show that Papnet is able to scale in a similar way as Pastry.



(a) Average node state (b) Number of hops

Figure 5.4: Comparison between Papnet and Pastry.

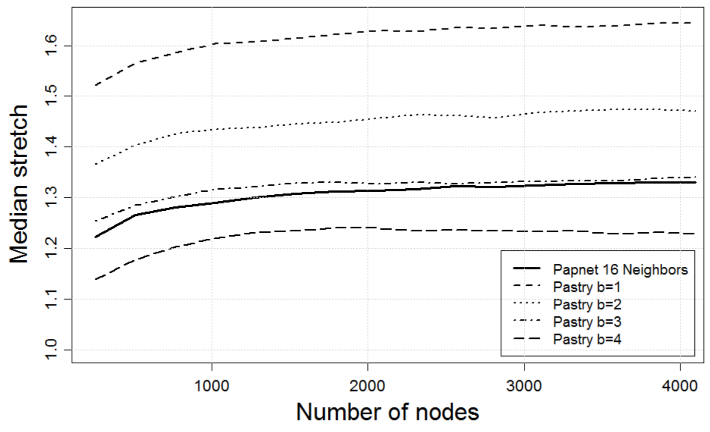


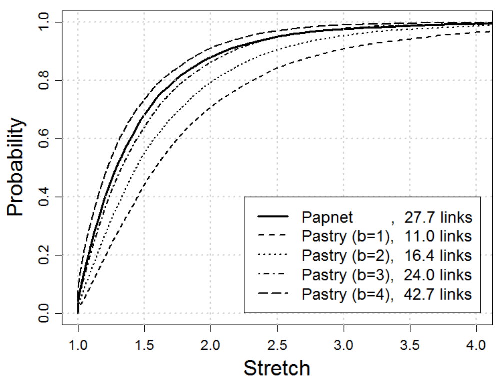
Figure 5.5: Median stretch at different network sizes

We evaluated the path latencies of both overlays by measuring the so-called *stretch*, which is the sum of latencies encountered on the path, divided by the direct latency between the first and last node on the path. Figure 5.5 illustrates the median stretch of $10,000$ test messages encountered at different network sizes. Note that the routing performance in terms of latency is better in Papnet than a Pastry network with $b = 3$, even though the actual routing paths are longer.

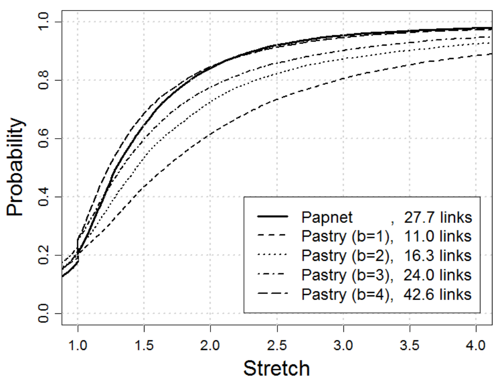
The Figures 5.4 and 5.5 show average values from 10 separate runs, measured in steps of 250 nodes. The shown tests were conducted using a simulated Euclidian latency metric. Additionally, we also made tests using real Internet latencies. In particular, we used the following two metrics:

Simulated latencies based on Euclidean coordinates
We assigned each node a random position in a two-dimensional coordinate system. The actual latency was calculated as the Euclidean distance between any two nodes coordinates.

Real latencies based on the King dataset
The King dataset [72] provides a matrix of real latencies between ca. 1200 real world DNS servers. Unfortunately, the King dataset is incomplete and approximately 6.6% of the matrix entries contain an invalid value of -1. For our tests, we selected $1,000$ nodes with a preference for those having the fewest invalid latency samples. We replaced any remaining entry where $L(x, y) = -1$ with its counterpart $L(y, x)$ if it was not invalid as well. All other invalid entries were replaced by $L(x, y) = L(x, z) + L(z, y)$ by looking up a node z so that the sum gets minimal.



(a) Simulated latencies



(b) King Dataset latencies

Figure 5.6: Latency stretch CDFs for n=1000.

Figure 5.6 shows the cumulative distribution functions (CDF) of the latency stretch of 10,000 test messages in a network of 1,000 nodes using either a) simulated or b) real latencies from the King dataset. In order to make Papnet and Pastry comparable, the average *node state* of each network has been measured and is depicted in the figure's legend.

Note that using real latencies, stretches less than 1.0 become possible. This is because the triangle inequality not necessarily holds in real world networks and thus an overlay path may be shorter than a direct connection.

The figure clearly shows that Papnet is able to provide low path latencies that are comparable to those of Pastry. In the case of simulated latencies, Papnet provides path latencies that are better than a Pastry network with $b = 3$ but worse than a network with $b = 4$. This seems feasible, since the average actual node state of the tested Papnet (27.7 unique links) lies in-between the states of the pastry networks with $b = 3$ and $b = 4$ (24.0 and 42.7 unique links).

Using real latencies from the King dataset, Papnet seems to provide less very low path latencies than the Pastry networks with $b \geq 3$. We suspect this to be caused by Papnet's average number of hops, which is higher than those of a Pastry network with $b \geq 3$ (see Figure 5.4b)). However, concerning probabilities higher than 0.85, Papnet seems to provide lower path latencies than any tested Pastry network.

Note that the comparison has been done using *optimal* routing tables for both, Papnet and Pastry. In Pastry, however, there is no guarantee that this *optimal* state will ever be reached.

Convergence to optimal state

Figure 5.7 demonstrates the performance of Papnets proximity optimization protocol over time for different network sizes. The horizontal axis represents time, measured in optimization steps. In each step, all nodes optimize exactly one level of routing links in a round-robin manner. The actual time between any two optimization steps can be freely adjusted, so that arbitrary tradeoffs between overhead and convergence speed are possible. In our tests we performed 1 optimization step each second and observed a constant message traffic of about 250 byte/s per node, caused by the optimization. This constant rate was observed in all of the tests and appears to be independent of the network size.

Notice how fast Papnet optimizes the links. In networks of up to 4,096 nodes, only 100 optimization steps suffice to let more than 60% of all routing-links become optimal. After 300 steps, more than 90% are optimal. Overall, the measurements indicate an eventual convergence against an optimality of 100%.

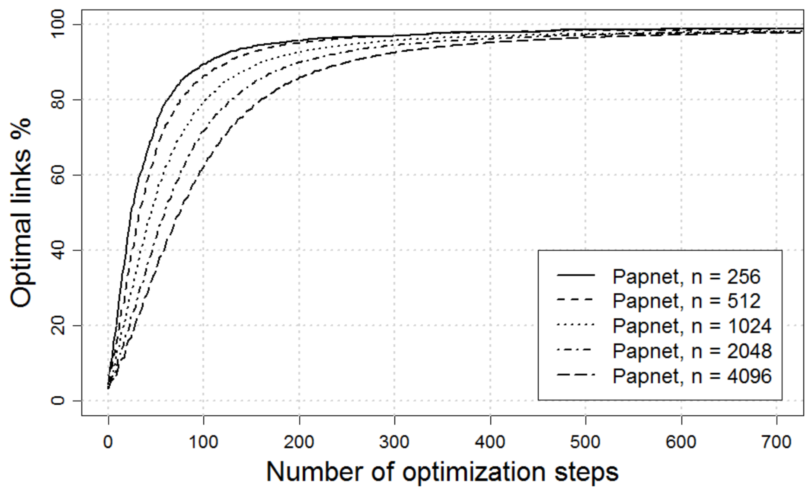


Figure 5.7: Optimality convergence

Effects of Latency Awareness

To demonstrate that latency awareness is crucial, we also compared Papnet to the non-latency-aware Overlay Chord. The properties of example networks of 10,000 nodes measured with our simulator[1] are given in Figure 5.8.

	Nodestate			Path Length			Latency Stretch		
	Min	Med	Max	Min	Med	Max	Min	Med	Max
Chord	12	15	18	1	8	14	1.0	7.62	894.11
Papnet	32	34	36	0	5	9	1.0	1.28	13.26

Figure 5.8: Chord vs. Papnet (10, 000 nodes).

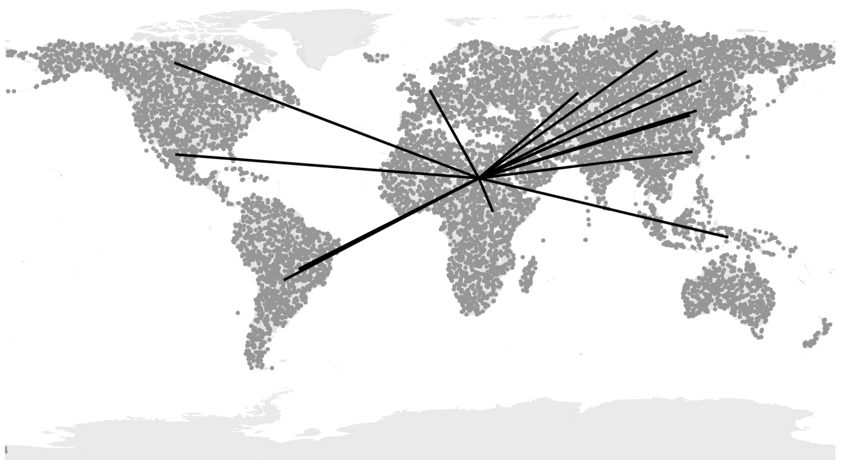
Figure 5.9 depicts the routing tables of example nodes in the simulated networks. As one can see, each network sets up a different number of routing links, which makes the different Overlays incomparable in principle. However, despite the different link count, one can observe that Papnet sets up more proximate links whereas Chords links exhibit a more uniform distribution.

Figure 5.10 illustrates the consequences of the latency awareness when routing messages. Chord is completely unaware of link latency and thus each routing step may result in large delays. Papnet on the other hand is latency-aware and induces only small delays with each step. While Papnet provides latency stretches as low as those in Pastry, only Papnet is able to *guarantee* an eventual convergence against the optimal state.

5.6 Conclusion

In this chapter, we have presented the proximity optimizations of Papnet. We have shown, that hash-free overlay networks are able to provide fast path latencies, comparable to those of hash-based overlay networks like Pastry. Furthermore, we presented a new algorithm that allows Papnet to guarantee eventual routing link optimality as well as fast optimality convergence. This guaranteed eventual optimality is a unique feature of Papnet and to our knowledge has not been accomplished in other Overlay networks yet, may they be hash-based or hash-free.

[1]Publically available online at http://www.papeer.net:8000/papnet/Papnet.html.


(a) Chord

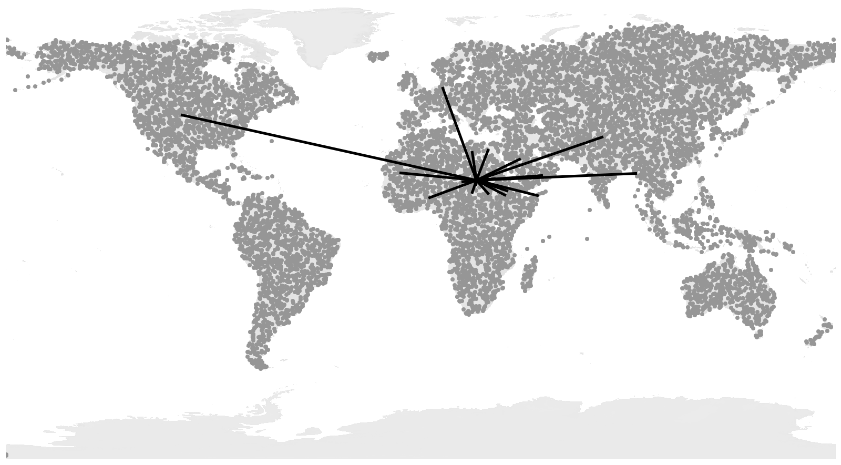
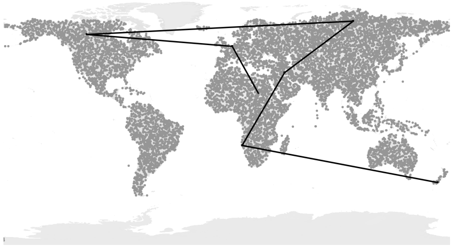
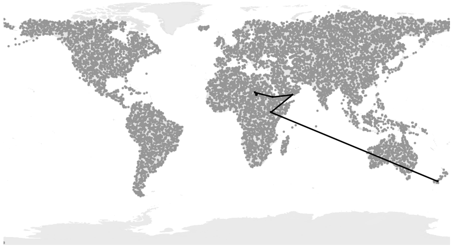
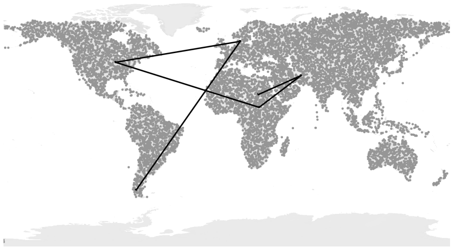
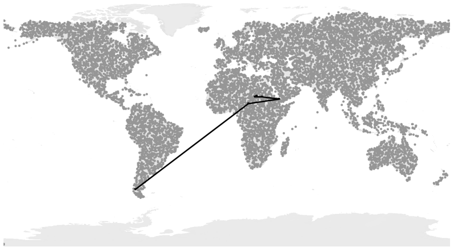

(b) Papnet

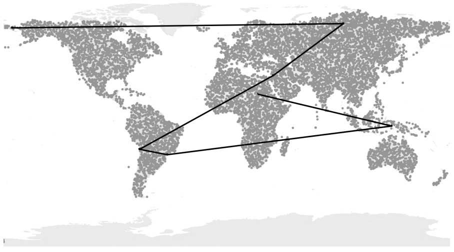
Figure 5.9: Example Routing-Tables in networks of 10, 000 nodes..

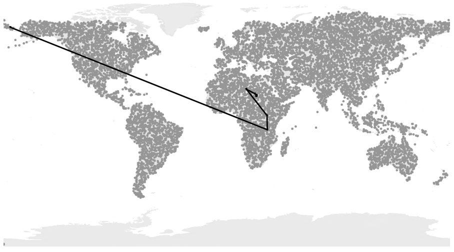

(a)

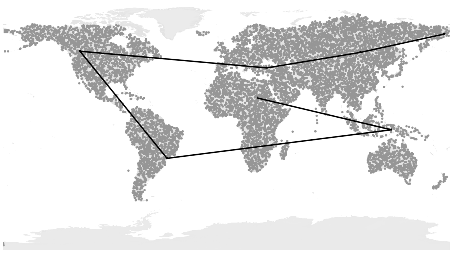

(b)


(c)


(d)


(e)


(f)


(g)

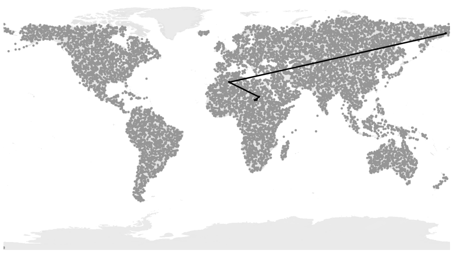

(h)

Figure 5.10: Example routes in Chord (left) and Papnet (Right).

6 Scalable spatial data processing with Papnet

Contents

In this chapter, we will present an application that greatly benefits from the properties of the Papnet P2P Overlay. In particular, we will use range queries and show how the constant global imbalance allows for a linear scaling with query load.

6.1 Introduction

The efficient processing of spatial data is a vast field of research that is being studied since many decades ago. Many centralized spatial structures have been developed to efficiently partition spatial search spaces and thus enable fast query evaluations. However, these structures hardly allow for distribution, since they often assume an initial knowledge of the total spatial data in order to construct the index or require a lot of rebalancing when dealing with dynamically changing data. Further, the need for top-down evaluation of these structures results in non-uniform load distributions in distributed settings.

We propose a distributed system that is solely based on the Hilbert curve, a so-called *space-filling curve* that is able to map any d-dimensional bound domain down to the unit interval $[0, 1)$. By utilizing the inherent space-partitioning structure of the Hilbert Curve, we demonstrate how approximations of spatial bounding rectangles can be computed in an ad-hoc manner that allow for an efficient evaluation. Our approach avoids the costly setup and maintenance of explicit data structures like R-Trees [73], while at the same time allowing for large-scale distribution and parallel query evaluation.

Further, we show how our approach can be implemented on top of a Peer-to-Peer (P2P) overlay that provides scalability and fault tolerance. Our approach differs from the usual mapping of spatial data to a *Distributed Hash Table* (DHT) that has been described several times in the literature in that we do not require any hashing or mapping layer that transforms data keys into entities suitable to be stored in a hash table. Instead, we utilize a p2p overlay that is capable of storing arbitrarily distributed data keys in a natural order and also allows for latency-optimized routing. Further, the P2P structure allows for the implementation of a simple but yet sophisticated and efficient load balancing strategy that guarantees a constant maximum data imbalance ratio throughout the distributed system.

In particular, our main contribution is the following: We show that a *space filling curve* such as the Hilbert curve **alone** provides sufficient space partitioning properties to allow for efficient evaluation of *window-* and *k-nearest-queries* in a distributed setting. To our knowledge, all other systems capable of processing such queries that have been described in the literature so far require the setup and maintenance of additional explicit spatial structures.

This chapter is structured in the following way: Section 6.2 gives an overview on related work. In Section 6.3 we describe our approach and how window- and k-nearest-queries can be evaluated in a distributed manner. Section 6.4 covers our prototype implementation and Section 6.5 presents measurements acquired using a real life data set. Section 6.6 concludes the chapter.

6.2 Related Work

Up to now, there exist a variety of approaches and publications on how to implement distributed spatial indexes using P2P architectures. There are basically two different paradigms [74]: (1) either a centralized spatial index (e.g. an R-Tree) is being partitioned and distributed, or (2) the multidimensional data may be mapped to fit the dimensionality of the P2P systems key space and is then being partitioned according to the topology and properties of the P2P system, e.g. a *distributed hash table* (DHT).

Systems of type 1 implement a distributed hierarchical spatial structure that needs to be evaluated in a top-down manner. Thus, the nodes that are located close to the root level of the spatial structure tend to get overloaded. Further, such index structures require costly rebalancing procedures upon node joins and leaves.

In [75], Mondal et al. propose a distributed spatial index of type 1 called *P2PR-Tree*. The index consists of rectangular blocks that are organized in a hierarchical set of levels and assumes that the top two levels are constructed statically. This requirement however casts doubts whether the index is able to handle arbitrary dynamic data.

Another index of type 1 is proposed by Tanin et al. in [76]. They propose a distributed quadtree structure and create a hierarchy of quadtree cells whose centroids are hashed to keys in a classic DHT P2P overlay. To tackle the problem of overloads and single-points-of-failure that arise from the need of queries being evaluated top-down from the root of the quadtree, they introduce an explicit concept to keep the top tree levels free of data.

Examples for approaches of type 2 are CAN [36] and Murk [7], which both use multidimensional P2P overlays. In [74], Kantere et al. propose a multidimensional spatial index of type 2 that can be implemented atop arbitrary DHTs. However, they require a static partitioning into regular grid cells, which can be identified by hashed values and therefore assigned to specific nodes in a DHT. Due to the regular grid structure, it remains questionable if the system is able to adapt to locally clustered data at arbitrary scale.

The probably closest approach to our own is SCRAP [77] which also uses a space filling curve to map spatial data to one dimension and then performs a partitioning of the linear space amongst nodes. To evaluate a window query, SCRAP calculates the curve segments relevant to the query (those that cover regions of the query window) and then evaluates the nodes responsible for these segments. The authors explicitly mention the usage of a query mapping technique that may produce false positives (curve segments not covered by the query window) and thus leads to evaluation of nodes that are actually not relevant to the query. The problem of k-nearest query processing is not addressed in the paper.

In contrast to SCRAP, we propose an iterative query evaluation technique that does not require an initial determination of all curve segments relevant to a query. Our technique avoids false positives as well as the processing of curve segments that do not contain any data. Further, we show how our approach is able to efficiently process k-nearest queries.

6.3 Spatial Partitioning

To simplify the problem of processing multidimensional spatial data, we make use of a so-called *space-filling curve* (SFC). Such curves are a class of bijective functions that map arbitrary dimensional but bound data down to the unit interval: SFC: $[0..1)^k \leftrightarrow [0..1)$. Note that this linearization of space does not induce any loss of information, since any bound multidimensional space contains just as many points as the unit interval $[0..1)$.

SFCs provide the nice property to preserve locality throughout the reduction of dimensionality. Spatial points that are close in the multidimensional domain are thus highly likely to also have close coordinates on the 1-dimensional curve. Thus, queries for arbitrary spatial areas are highly likely to require only the evaluation of small subsets of these partitions. Another key advantage of this approach is that a single linear data domain ideally allows for non-complex partitioning and distribution.

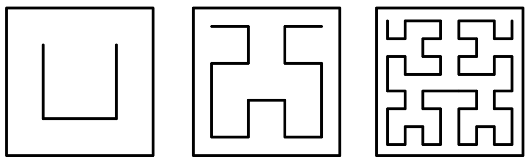


Figure 6.1: Hilbert curves of first, second and third order.

In particular, we use the Hilbert Curve (HC) which is depicted in Fig. 6.1. We prefer it over other SFCs, because it has been shown [78] to provide much better locality properties than other known SFCs, e.g. the Z-Curve [79]. The Hilbert Curve is defined in terms of *orders*, where each order can be constructed based on the previous lower order in a recursive manner using the following (informal) rule:

- Divide curve into 4 equal-sized copies.
- Rotate the start quarter counter-clockwise.
- Rotate the end quarter clockwise.
- Connect appropriately.

Figure 6.2 illustrates the above transformation.

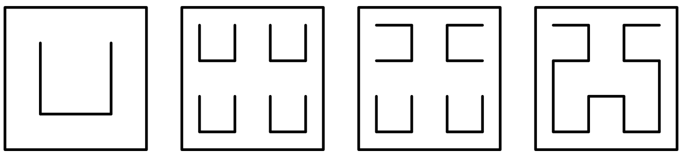


Figure 6.2: Hilbert Curve construction step.

The Hilbert Curve is a fractal curve that assigns each spatial point a unique one-dimensional coordinate (Hilbert value/path) and vice versa. This linear coordinate does not necessarily need to have a finite representation, i.e. the point $x = 0.5$, $y = 0$ gets approximated by the curve as the orders increase, but there is no finite Hilbert value that represents the corresponding curve point. However, given that we describe and process spatial coordinates only by variables of fixed precision (i.e. 64 bit *doubles*), we can assign *finite* Hilbert values to all such spatial points.

Using an SFC for reduction of dimensionality, we gain a natural order on any set of spatial points. Figure 6.3 gives an example mapping, where 2500 Points (left) are mapped using a Hilbert Curve (middle, showing a lower order for illustration) resulting in an ordered set of spatial points (right, connected).

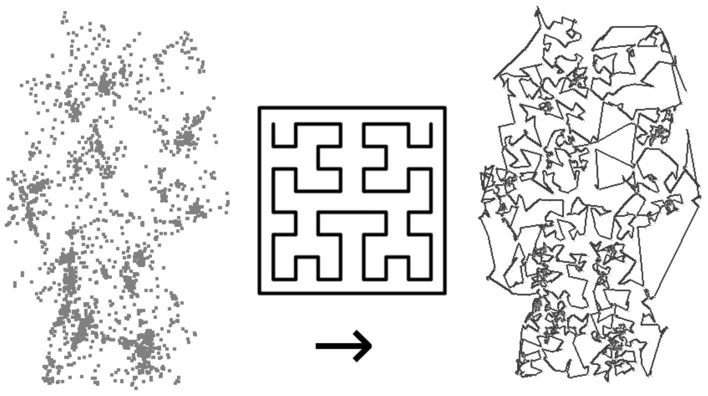


Figure 6.3: Mapping spatial data to a Hilbert curve.

Any ordered set of points on the Hilbert Curve is embedded into the unit interval [0..1). This interval may be thought of as forming a key space with ring topology, which can be partitioned into (nearly-) equal-sized segments, either by key range or element count. We prefer the latter, because we want to distribute the total data to autonomous nodes that shall manage equal shares. An example partitioning can be seen in Figure 6.4.

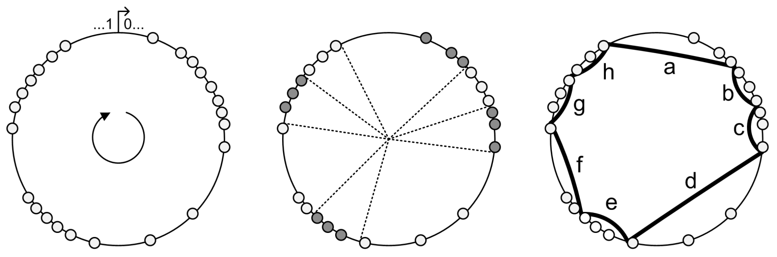


Figure 6.4: Partitioning a mapped set of spatial points by element count.

The single segments of the partition can largely differ in key space size, because there may exist clusters of data keys. This is because real world data is not likely to be distributed uniformly across the spatial dimensions and the curve mapping preserves clusters to a large degree. This effect can be visualized by mapping the segment boundaries back to spatial coordinates as depicted in Figure 6.5 (the labels *a*-*h* are only for illustration and do not exactly resemble those in Fig. 6.4). As the number of segments is increased, densely populated spatial areas will contain a larger fraction of segments.

Since the curve mapping preserves locality, queries for rectangular windows will only need to evaluate highly co-located segments. An example query for the area around Berlin/Germany is shown in Figure 6.5 (right). This query only affects the highlighted 3 of the total 64 segments. Note that there is indeed a third segment in the center which is barely visible, because it only covers the small-in-area but densely populated center of the German capital.

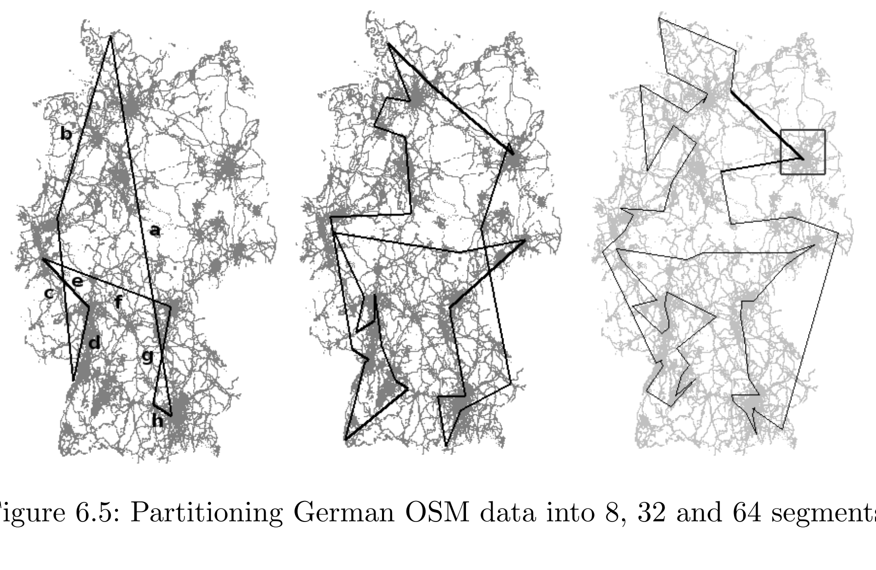


Figure 6.5: Partitioning German OSM data into 8, 32 and 64 segments

Quad Tree Structure and Window-Queries

The Hilbert Curve is by definition a structure that partitions space into regular subspaces. Each partition step can be identified by a binary string of length d, where d is the dimensionality of the spatial source domain. If $d = 2$, each step is from the set $S = \{00, 01, 10, 11\}$. Figure 6.6a) illustrates this quad tree structure.

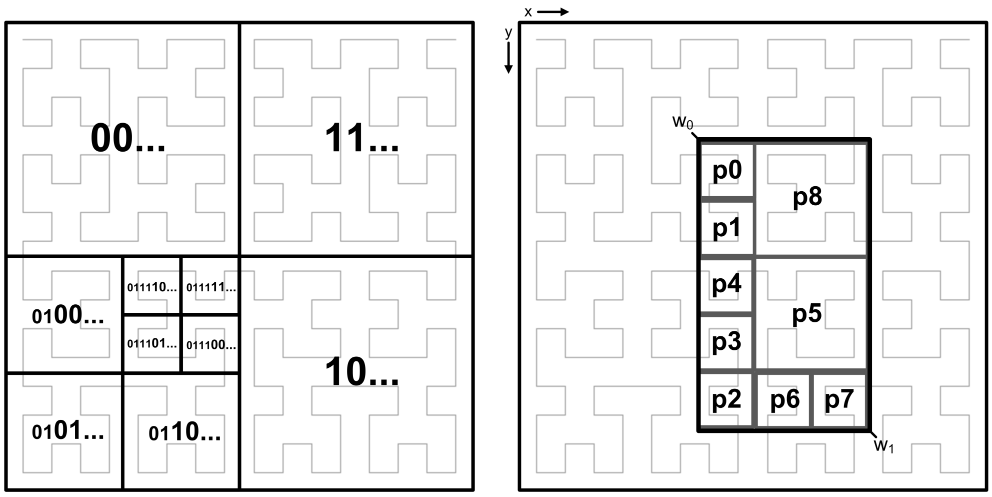


Figure 6.6: a) Quadtree structure. b) Example window query

We call the set of sequences of such steps the Hilbert paths $P = S^*$. Each path $p \in P$ identifies a unique spatial cell in the source domain. Formally, any particular paths p and p' may be concatenated to form another path $q := p \cdot p'$. Paths also allow for comparison using a lexicographical order, e.g. $01 > 00$ and $1100 > 11$. Queries for arbitrary rectangular spatial areas (so-called *window queries*) can be evaluated iteratively by traversing this quadtree structure. One way to do this, is to calculate a set of disjunctive quad tree cells that entirely cover the spatial query window and then evaluate them along the curve. Figure 6.6 b) shows an example evaluation of the query-window $w_0 = (0.375, 0.25)$, $w_1 = (0.75, 0.875)$.

To find all relevant spatial cells in order we can use Algorithm 12, which takes 4 parameters: the current Hilbert path p, the minimum Hilbert path $p_{\min}$ and the corners of the query window w_0 and w_1. The current path p is initially empty, gets extended by two bits in each recursion and ultimately identifies the resulting spatial cell we are looking for. The minimum Hilbert path $p_{\min}$ is used to exclude cells that we already processed, i.e. if $p_{\min} = 10$ then we are only interested in the second half of the Hilbert curve. The algorithm returns the path of the next cell to process, or *null* in case there is none left.

Algorithm 12: P nextCell(p, $p_{\min} \in P$, $w_0, w_1 \in [0..1)^2$)

```
// test if current cell contains cells $\geq p_{min}$
if $\nexists p' : p \cdot p' \geq p_{min}$ then return null
// test for window intersection
Let $c_0, c_1 \in [0..1)^2$ := spatialCorners($p$)
if intersect($c_0$, $c_1$, $w_0$, $w_1$) $= \emptyset$ then return null
// test if cell is entirely contained in window
if $p \geq p_{min} \wedge (c_0, c_1) \subseteq (w_0, w_1)$ then return $p$
foreach $d \in \{00, 01, 10, 11\}$ do
    Let $p_{\text{sub}}$ := nextCell($p \cdot d$, $p_{\min}$, $w_0, w_1$)
    if $p_{sub} \neq$ null then return $p_{sub}$
return null
```

Algorithm 12 first checks whether there exists any path suffix p' that can be concatinated to the current path p to form a path that is greater that $p_{\min}$ (**line 2**). If, for example, $p = 00$ (upper left quarter of the Hilbert curve, cf. Fig 6.6) and $p_{min} = 10$, then obviously there is no path with prefix 00 that is greater than $p_{\min}$ (meaning that the upper left quarter of the curve does not contain any cells that belong to the second half of the Hilbert curve). Next, we calculate the spatial minimum and maximum corner c_0, c_1 of the cell defined by the current path p (**line 4**) and check if the cell intersects with the query window (**line 5**), e.g. in Fig. 6.6 the cell $p = 00$ has corners $c_0 = (0, 0)$, $c_1 = (0.5, 0.5)$ and intersects with the query window. Then, we test if the current cell is entirely contained in the query window and if its path is not lower than $p_{\min}$ (**line 7**). If both can be confirmed, we found the next cell. Otherwise, we will recurs (**lines 8-10**) and search all sub-cells of the current cell by concatenating 00, 01, 10, 11 to the current path p (in exactly that order). If no matching cell was found, a *null* value is returned (**line 11**).

Using Algorithm 12, the window query depicted in Figure 6.6a) expands to:

$$\begin{gathered}
\text{p0} := \text{nextCell}(\emptyset,\ \emptyset,\ w_0,\ w_1), \\
\text{p1} := \text{nextCell}(\emptyset,\ \text{inc(p0)},\ w_0,\ w_1), \\
\text{p2} := \text{nextCell}(\emptyset,\ \text{inc(p1)},\ w_0,\ w_1), \\
\dots\ , \\
\text{p9} := \text{nextCell}(\emptyset,\ \text{inc(p8)},\ w_0,\ w_1) = \textit{null}.
\end{gathered}$$

Note how we raise the minimum path $p_{\min}$ with each iteration by incrementing the path of the previously processed cell. The inc(p) operation looks up the rightmost step $s < 11$ in p, cuts all following steps and then increments, e.g. inc(00) = 01, inc(0101) = 0110, inc(0011) = 01, inc(110011) = 1101, inc(1111) = *null*.

Algorithm 13: windowQuery($w_0, w_1 \in [0..1)^2$, data $\subset P$)

```
1  Let results $\subset P := \emptyset$    // result set
2  Let $i \in \mathbb{N} := 0$    // data index
3  while $i < |data|$ do
4      // test if data element is in query window
5      if notInWindow($w_0$, $w_1$, coordinate(data[i])) then
6          // calculate next cell
7          Let $p \in P :=$ nextCell($\emptyset$, data[i], $w_0$, $w_1$)
8          // get index of first element $\geq p$
9          $i :=$ data.ceilingIndex($p$)
10     else  results += data[$i$++]
```

Of course there are lots of spatial query windows where an evaluation as described above results in large numbers of cells. The number of cells can however be bound by the number of actually existing data elements, simply by skipping empty cells. Algorithm 13 efficiently evaluates a window query. The algorithm takes the query window (w_0, w_1) and a data set (e.g. a nodes partition) as input. The data set is a sorted list of ascending Hilbert values representing the spatial positions of data objects, which is being iterated using an index counter (**lines 2-3**). In each iteration, the current data object will be tested against the query window (**line 5**) and being added to the result set (**line 10**) if it is contained (increasing the index counter afterwards). If not, then the Hilbert path of the next spatial cell where the Hilbert Curve re-enters the query window will be calculated using the current objects Hilbert value as minimum path (**line 7**). Finally, we increase the index counter to point to the next data element whose Hilbert values is at least as high as the value of the calculated cell (**line 9**).

Assuming the data space has been partitioned into n partitions $R_1 \cdots R_n$, the window query algorithm easily extends to a distributed evaluation. After calculating the first query cell $p_0 =$ nextCell($\emptyset$, $\emptyset$, w_0, w_1), the query is forwarded to the node responsible for the partition R_x that p_0 lies in: $p_0 \in (R_{x-1}, R_x]$. This node will then process its local data (using Algorithm 13 and forward the query afterwards to the node being responsible for $p_1 :=$ nextCell($\emptyset$, inc(R_x) , w_0, w_1). The forwarding continues until the entire query window has been evaluated.

We thus have defined a sequential processing order for window queries that can be implemented in a distributed system. Note that while the traversal order is sequential, the actual data processing can be parallelized by postponing local data processing and forwarding the query first. This parallelization will pay off whenever the local evaluation is expensive, e.g. when processing *k-nearest* queries.

Remark: We think that even the node traversal itself does not necessarily need to be strongly sequential, but can be efficiently parallelized by using local knowledge about the global partition provided by the underlying network structure and load balancing technique. In our case, each Papnet [71] node (see Section 6.4) sets up links in exponentially increasing node distances, i.e. each node knows the 2^kth node for $k \geq 0$ in both directions on the ring. Further, our used load balancing technique ensures a *constant* maximum data load imbalance. We are thus able to estimate the total data amount stored between two linear addresses. Now the key idea is to 1) calculate the first and last cell relevant to the query, respectively their corresponding minimum and maximum Hilbert value, 2) estimate the amount of data elements between these values 3) estimate the node that is responsible for the median element 4) send the query to both nodes being responsible for the minimum and median value and 5) repeat the procedure at those nodes. However, this strategy requires further investigation and we leave it as future work.

K-Nearest Queries

Provided with an efficient technique to evaluate window-queries, we will now show how to process so-called *k-nearest* queries (knn), which means to find the k nearest points to a given query pivot. Our technique uses the assumption that a query window is provided which bounds the spatial area to search. It extends to an assumption-free algorithm by starting with a small window and repeated execution with a larger window in case less than k points have been found.

The classic approach to efficiently calculate the k closest entities to a given query pivot is to utilize an explicit spatial data structure from a family called *Regional Trees* or R-Trees [73]. An R-Tree clusters groups of close nodes within *minimum bounding rectangles* (MBR) and assembles a hierarchical structure of enclosed MBRs. At each tree node, an R-Tree can be efficiently searched in a depth-first manner by calculating the minimum distance (query pivot to MBR) for each child and processing them according to proximity. During the search, most nodes can eventually be pruned (as soon as k data elements of closer distance are found).

R-Trees however require costly maintenance and rebalancing operations due to dynamic inserts and deletions of data. Especially in a distributed setting, this can result in complex message exchange protocols. Further, R-Trees are designed to be evaluated in a top-down manner, which is fine for a centralized system, but results in overloaded nodes in a distributed system – those that are located close to the root level of the tree.

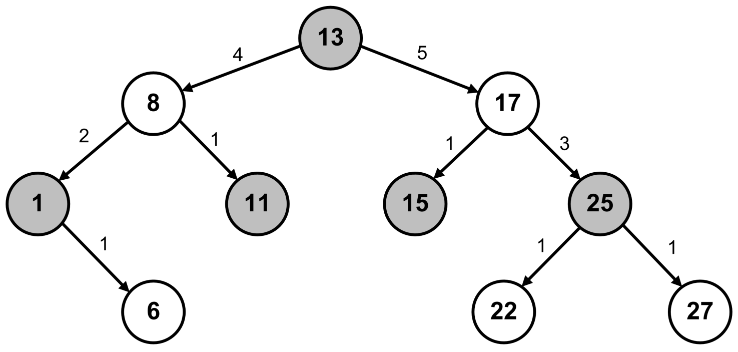


Figure 6.7: Augmented Red/Black-Tree

We take a different approach and require nothing but a simple *red/black tree* data structure to store the Hilbert values of the spatial entities. This structure provides a worst case complexity of $O(\log n)$ for search, insert and delete operations. Additionally, we keep track of the cardinality of the subtree below each tree node. This extended structure is known as the *Augmented* Red/Black-Tree (cf. [80]) and can be seen in Figure 6.7. Note, that keeping track of subtree cardinalities is only a simple extension which does not alter the complexities of the trees operations.

The augmented tree structure allows us to efficiently retrieve elements of arbitrary rank, especially the median element of arbitrary element ranges with known boundary ranks. This in turn enables us to perform an efficient on-the-fly binary partitioning of the data stored at a node, whenever a k-nearest query is being evaluated.

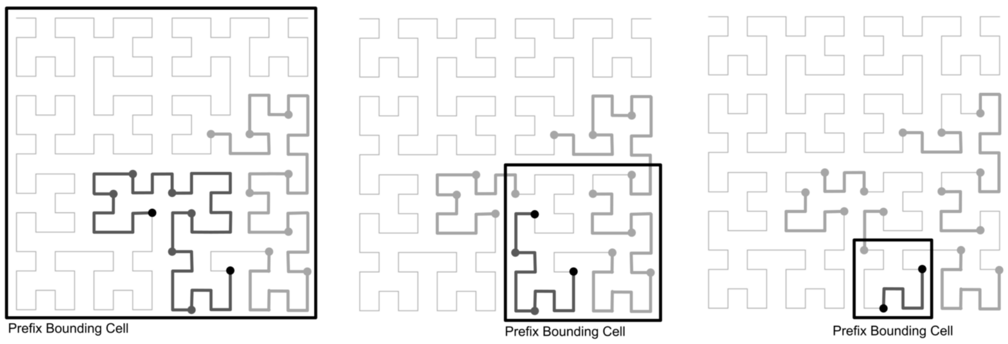


Figure 6.8: PBC Calculation throughout dataset halving

The second ingredient to our strategy is the on-the-fly approximation of *minimum bounding rectangles* just as they are explicitly constructed in R-Trees. This is achieved by calculating the smallest cell of the Hilbert Quadtree that entirely covers the elements of a specific range of data. We call this minimum cell the *prefix bounding cell* (PBC).

Figure 6.8 illustrates the PBCs of a data set that is being binary partitioned. Note that the PBC size is not necessarily lowered in *each* halving step, i.e. the PCS of half the data (Fig. 6.8 left) and the total data (not shown) are identical. However, as we will explain later, *some* of the PBCs must shrink.

The PBCs allow us to utilize the very same search algorithm usually applied to R-Trees to efficiently calculate the k nearest points. In particular, we keep a priority queue of PBCs that is sorted by minimum distance to the query pivot and process the closest PBC first. Initially, the queue only contains the root PBC that comprises all of a nodes data. Whenever we process a PBC that contains more than a threshold element count t, we determine the median element of that PBC, split the data, calculate the two child PBCs and reinsert them into the queue. Note that each time a PBC is split, there is at least one child PBC where all elements share a common Hilbert prefix that is at least one bit longer than in the initial PBC. Therefore, if we split any arbitrary PBC into two children and then split those again, we end up with a set of 4 PBCs where at least 1 has a smaller side length than the initial PBC. Since we are splitting using the median element, all of the 4 PBCs contain $\frac{1}{4}$ of the data elements contained in the initial PBC.

Figure 6.9 shows the evaluation of an example query for the k nearest entities to a query pivot. You can see the PBCs and their corresponding distance to the pivot for a partitioning using a threshold of $t = 2$ (max. elements in a PBC).

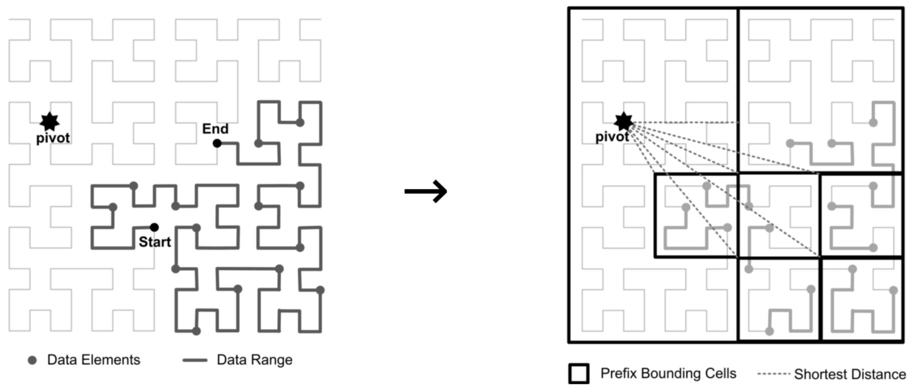


Figure 6.9: Query evaluation with distances to PBCs

Time Complexity

To analyze the complexity of our knn-query evaluation approach, we will consider the reduced problem of finding the single nearest data point to a given query pivot. The worst case complexity of such a nearest-point query in a spatial tree structure of dimension d is $O(d \cdot n^{1-\frac{1}{d}})$ [81]. The average case is however of lower complexity. We will now try to give an intuitive justification on why our approach is able to provide at least an average complexity of $O(\log^2 n)$ for nearest-point queries.

First, we assume the existence of an ideal guessing function $g(A, B)$ that is able to decide which one of two disjoint data sets associated with spatial areas A and B contains the closest point to the query pivot. Now we take the PBC that contains the entire data set and perform a repeated binary partitioning. In each step we split the current PBC, apply g to both children, pick the one that contains the closest point and discard the other. Since each partitioning halves the set of data, it is easy to see that we will find the desired nearest point after $\log_2 n$ steps.

Of course, in reality no such ideal guessing function g exists. What however does exist, is the observation that given two areas A and B, the closest point is more likely to be in the cell having the smaller minimum distance to the query pivot. We assume the existence of such a (more realistic) guessing function $g'(A, B)$ which, provided with two *distinct* areas A and B, is able to correctly decide based on the minimum distance of the pivot to A and B. If $A = B$ then there is no candidate g' can decide on, since both are of equal distance to the pivot. We thus need to show that our partitioning method will provide *distinct* areas as suitable input to g'.

A single PBC that is split twice results in 4 PBCs that contain $\frac{1}{4}$ data points each and at least one of them covers an area that comprises only $\frac{1}{4}$ of the initial PBC area or less. Let A be exactly this smaller PBC and B be the smallest PBC that contains the three remaining PBCs. Since A and B must be distinct and we assume $g'(A, B)$ to guess correctly, we can thus prune at least $\frac{1}{4}$ of the search space: either A or B. In the worst case, only $\frac{1}{4}$ of the data can be discarded while $\frac{3}{4}$ remain. The time needed for one iteration (partitioning) thus calculates to

$$T(n) = 3 \cdot T(\tfrac{n}{4}) + c$$

which results in a complexity of $O(\log n)$. Note that so far, we have ignored any processing costs but those of the partitioning. Each split of a PCB requires a lookup of the median element of its data set and also the priority queue of PCBs needs to be maintained. The complexity of these operations is at most in $O(\log n)$. We can thus estimate a processing complexity of $O(\log^2 n)$ for an average nearest point search (given that g' guesses right).

6.4 Implementation

The methods described in the previous chapters may be implemented using a classic master-slave architecture, where a central site manages the space partition. However, for fault-tolerance and scalability reasons, we favor a fully distributed implementation based on a peer-to-peer (P2P) architecture.

P2P Overlay

Our approach requires an overlay network that allows for both data and nodes to be non-uniformly distributed in key space. Classic P2P-Overlays like Chord [2] and Pastry [3] do not fulfill these criteria – their routing structure suffers severely from non-uniform distributions as we have shown in [82]. There are, however, alternative overlays like SkipNet [5], Chord# [6] and Papnet [71], that are able to handle non-uniform key distributions while providing the same routing and space complexities as classic DHTs. Out of those, we prefer our own development Papnet, because of its latency-optimized routing abilities and load-balancing capabilities.

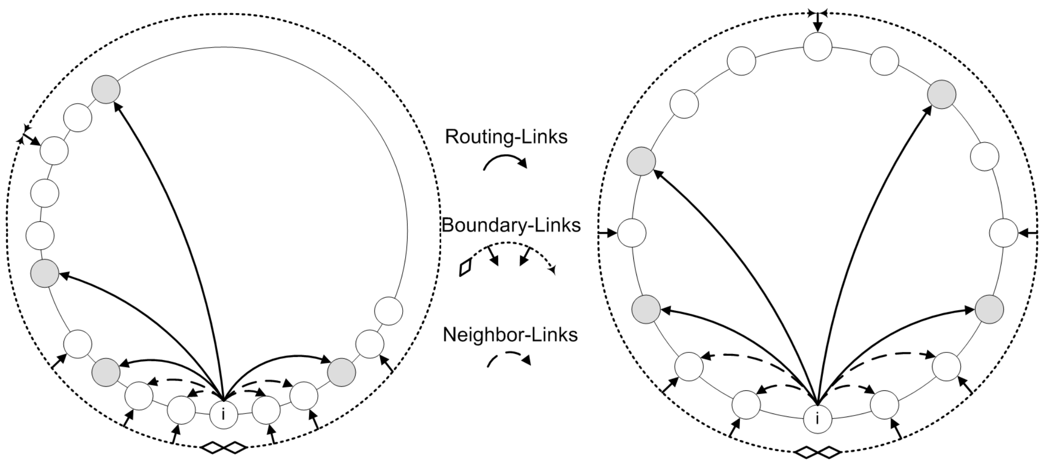


Figure 6.10: Key space and node space view of a Papnet

Figure 6.10 illustrates the network views of a single Papnet node: The key space view (left) which allows for arbitrary skewed key distributions and the logical view (right) that enables efficient routing without assumptions on the key distribution. There are three types of links: 1) Neighbor-Links that ensure network connectivity, 2) Boundary links enabling routing in O(log n) and 3) Routing links that allow for latency-optimized routing. A more detailed description of Papnet was given in chapters 4 and 5.

Balancing the partitions

So far, we have assumed that all partitions manage an equal share of the global data. Keeping such a uniform load distribution in a distributed setting can require many load balancing operations which is often not feasible. There are however load balancing techniques, that are able to maintain very good load distributions with only constant amortized costs for insertion and deletion. One such technique is the *On-Line Load Balancing of range-partitioned data* (OLB), proposed by Ganesan et al. [7], which ensures a global constant imbalance ratio of most- to least-loaded node. While the OLB cannot be applied to classic DHT-like overlays, hash-free overlays such as Papnet naturally allow for an integration (as shown in [71]).

6.5 Evaluation

We evaluated the performance of our solution on a cluster consisting of eight machines, each equipped with two Intel Xeon 5355 CPUs running at 2.66GHz, 32GB of RAM and interconnected by Gigabit Ethernet. Each CPU had 4 cores so that we were able to deploy up to 64 worker instances, each assigned to a single exclusive core. A ninth identical machine was used to generate the workload for our tests.

In our tests we used real world data from the publically available *Open Street Map* project[1] (OSM). Since this data set is very large (> 1 Billion objects), we only used the extract of Germany[2] (ca. 70 Million objects). To avoid long data stage-ins (and thus valuable computing time on the cluster) we ran our tests using a random sub-extract of 1 Million objects.

We tested our system by issuing 5,000 * size(network) queries for the 10 nearest objects around a pivot location. The pivot locations were selected randomly from the set of 1 Million objects. This selection ensures a realistic query distribution in that densely populated areas will be queried far more often than lightly populated ones. To maximize the throughput, we ran 20 queries in parallel and set the PBC split threshold to $t = 200$ (see Section 6.3).

[1]http://www.openstreetmap.org
[2]http://download.geofabrik.de/osm/europe/germany.osm.bz2

Data Distribution

In order to provide scalability we require a very good data partitioning that assigns each node an almost equal share of the global data load. We greatly benefit from the Ganesan online load balancing [7], which guarantees a constant load imbalance ratio, even upon dynamic data changes.

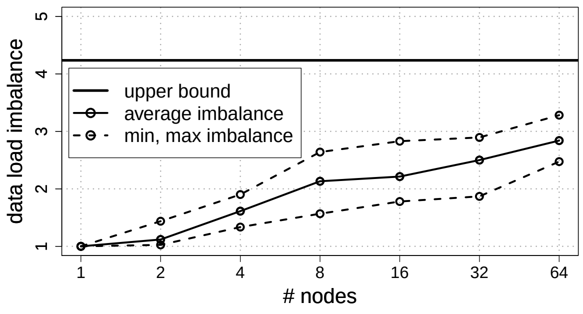


Figure 6.11: Data imbalance ratio

Figure 6.11 shows the imbalance ratios we encountered at different network sizes throughout all of our tests. As the number of nodes increases, the ratio rises due to the fact that a larger quantity of partitions is more likely to differ in size. However, the imbalance does and will not cross the upper maximum bound shown and proven in [7].

The constant imbalance ratio results in a very good data load distribution and is the basis for the scalability of our approach. In particular, we encountered a highly stable mean load per node, which can be seen in Figure 6.12 for different total amounts of data. Notice how the mean load exhibits a perfect linear decrease with increasing node count.

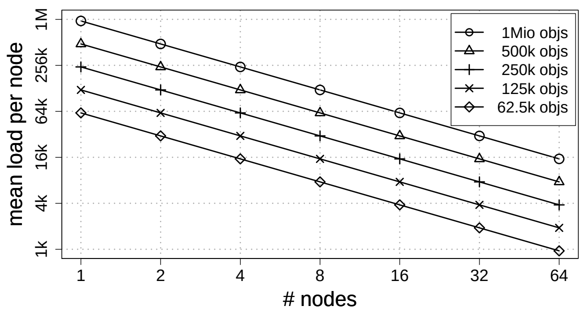


Figure 6.12: Mean data load

Processing Costs

As we explained in section 6.3 there are two ways to evaluate a query: either sequentially, with each node passing the query to the next node after performing a local k-nearest search, or in parallel with all nodes performing a local search at nearly the same time. The former has the advantage that intermediate results are passed from node to node and can thus be used for an iterative reduction of the search space, while in a parallel evaluation, each node performs a local search without external knowledge. Thus in parallel evaluation – which we used throughout our tests – the number of nodes relevant to a query can affect the total number of distance calculations that need be performed throughout the system.

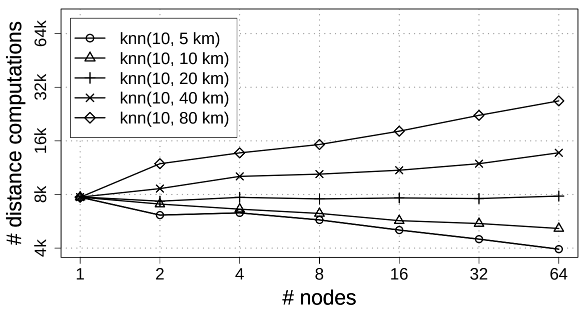


Figure 6.13: Total distance calculations

This impact can be seen in Figure 6.13, where queries for radii $\geq$ 20 km induce rising numbers of distance calculations with increasing node count. Note that the figure shows a total number of distance calculations, whereas the costs *per node* are much lower, as can be seen in Fig. 6.14. Again, an asymptotic linear decrease is noticeable.

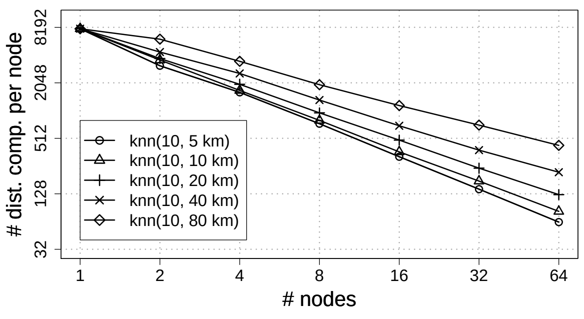


Figure 6.14: Distance calculations per node

Query Latencies and Locality

The benefit of parallel query processing is fast *round trip times* (RTT), i.e. the time passing from submission of a query until its full evaluation.

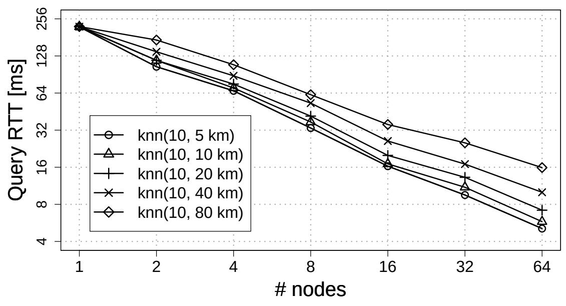


Figure 6.15: Query RTTs

Figure 6.15 shows the RTTs encountered in our tests. Since we ran multiple queries in parallel, the processing queues quickly fill up in low network sizes. As the number of nodes increases, a query is more and more likely to be immediately processed instead of being queued, so that response times ≤5ms can be reached.

Figure 6.16 shows the average number of nodes visited by a query. To allow for global scaling, each query should require an evaluation on as few nodes as possible. The number of nodes relevant for a query depends on two parameters: the query radius and the network size. As the system size is increased, the graphs seem to converge towards a fixed percentage of nodes. We presume the relative spatial area queried to be a lower bound of the convergence value, which in case of Germany (area $\approx 357,111\text{km}^2$) calculates to $\approx 0.044\%, 0.18\%, 0.70\%, 2.8\%$ and 11.3% for radii 5-80 km. The non-uniform distributions of both data and query pivots lead to an increased convergence percentage.

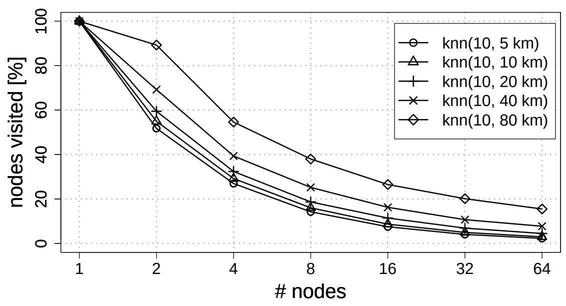


Figure 6.16: Percentage of nodes visited per query

Throughput and Efficiency

Last but not least, we present throughput measurements acquired with our prototype. We calculated the throughput in *queries per second* by dividing the number of queries by the time it took for all queries to complete. As in all of our other tests, the total amount of data stored in the system was 1,000,000 records.

#nodes:	1	2	4	8	16	32	64
knn(10, 5km)	91	191	300	601	1224	2101	3928
knn(10, 10km)	91	170	285	537	1175	1820	3453
knn(10, 20km)	90	170	264	485	1006	1510	2788
knn(10, 40km)	90	145	227	381	766	1178	1996
knn(10, 80km)	90	116	184	323	563	792	1236

Figure 6.17: Queries per second for different radii

Figure 6.17 shows the average resulting throughput encountered in 3 runs for 5 different radii. The corresponding graphs are depicted in Figure 6.18. It turns out that our approach scales almost linear with the number of nodes in the system.

It does not scale *exactly* linear because the further the system size is increased, the larger the set of nodes managing the objects in the area of interest will get. Because of the steadily increasing number of visited nodes, the communication overhead will eventually dominate the gain in processing speed resulting from the parallelization of query evaluation. However, the larger the problem space (total amount of data in the system), the later this effect will be noticeable when scaling the system.

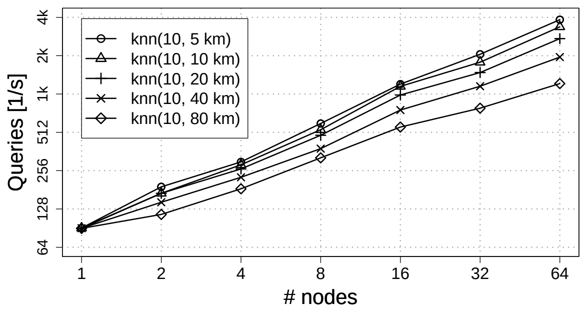


Figure 6.18: Throughput by network size

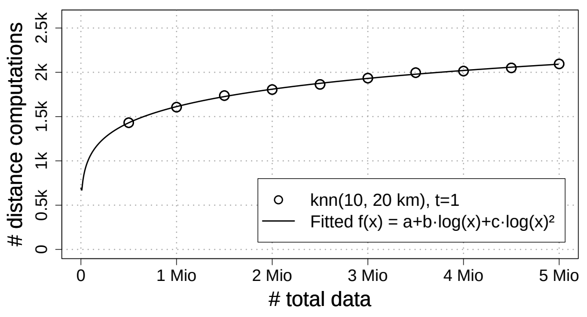


Figure 6.19: Processing costs by data amount

Finally, we evaluate the scalability of our approach in terms of data count. To get unbiased results, we ran this test separately on a single node and reduced the split-threshold t to 1, so that cells get split until they contain only a single element (see Section 6.3). Fig. 6.19 shows the resulting average processing costs of single queries for different total amounts of data. Clearly, a sublinear growth of processing costs with increasing data amount can be observed, which fits a function $y = a + b \cdot \log x + c \cdot \log^2 x$ (also depicted) with a coefficient of determination of $R^2 = 0.998$. Therefore, the results confirm our average time complexity estimation of $O(\log^2 n)$ (see Section 6.3).

6.6 Conclusion

In this chapter, we presented a novel strategy for distributed processing of spatial point data. It is to our knowledge the first such technique that avoids the construction (and maintenance) of an explicit spatial data structure. Using only the implicit spatial quad-tree that is inherent to the 2D Hilbert Curve, we have shown how to efficiently process the two most important spatial query types: window- and k-nearest-queries.

Our evaluation based on real world data and realtime processing has shown our solution to scale almost linearly with the number of processing nodes in the system. Further, we have shown that our system can be implemented on top of a decentralized P2P system that provides scalability and fault tolerance. In contrast to other P2P approaches, our solution is not based on a DHT and does not require hashing at all.

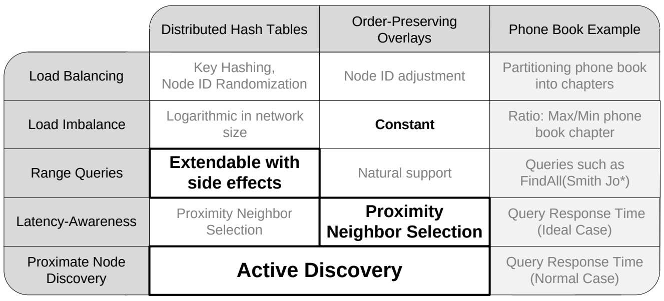
– This page has been intentionally left blank –

7 Conclusion

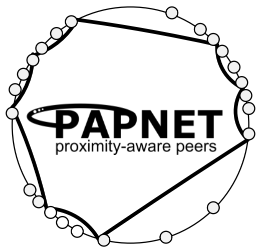
This thesis presented the new P2P Overlay network Papnet. It is one of the few Overlay networks that are able to store keys in a natural order and do not require any Hash mapping of keys. The natural order broadens the field of applications, going beyond the capabilities of current DHT Overlays by introducing support for range queries. As an example application, we demonstrated that a distributed RDF store can greatly benefit from such a natural order to perform join queries and avoid load hotspots.

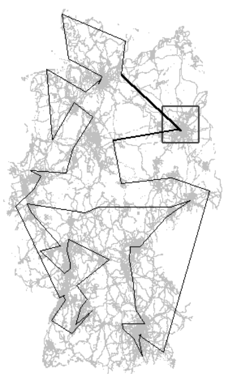
The natural order stabilizes the scaling behavior of the global system by allowing for load balancing with guaranteed constant imbalance. We have shown that classic DHT systems too can be extended to support this kind of load balancing, as well as range queries, but their topologies rely on a restricted finite id space and degenerate under worst case conditions, e.g. extremely skewed data and node distributions. The cooperative topology creation schema of Papnet, however, is well-suited for *any* data and node distribution since its structure is independent of the actual key space.

In contrast to other order-preserving P2P Overlays, Papnet is latency-aware. We have shown that Papnet is able to provide a constant path stretch of only 2 and guarantees a fast convergence against that value. The path latency is independent of the actual network size – a property that makes Papnet superior to any existing order-preserving P2P Overlay. The convergence guarantees have been confirmed using simulated as well as real internet latency data and the scalability of our approach has been demonstrated for up to $50,000$ nodes.



	Distributed Hash Tables	Order-Preserving Overlays	Phone Book Example
Load Balancing	Key Hashing, Node ID Randomization	Node ID adjustment	Partitioning phone book into chapters
Load Imbalance	Logarithmic in network size	**Constant**	Ratio: Max/Min phone book chapter
Range Queries	**Extendable with side effects**	Natural support	Queries such as FindAll(Smith Jo*)
Latency-Awareness	Proximity Neighbor Selection	**Proximity Neighbor Selection**	Query Response Time (Ideal Case)
Proximate Node Discovery	**Active Discovery**		Query Response Time (Normal Case)





Further, we proposed a new algorithm for processing distributed window- and k-nearest-neighbor queries on spatial point data. Using Papnet as the core routing layer, the new algorithm greatly benefits from ordered keys and the constant data imbalance guarantees, which result in an asymptotically linear scalability of throughput with node count. The latency optimizations of Papnet ensure it can be deployed in large clusters without causing severe routing overhead or packet congestion. We have shown that the system provides short query response times and query load scalability, especially for compute-intensive computations such as k-nearest-neighbor queries. The proposed algorithm is superior to existing solutions in that it can handle arbitrarily skewed spatial data and does not require the setup and maintenance of an explicit spatial data structure.

Papnet has been successfully deployed and tested in local environments, scaling up to 50,000 nodes on 100 physical machines, but it still needs to be evaluated in true large-scale and real-life settings. Further, since it strongly relies on node cooperation, Papnet is currently only suitable for entirely controlled and trusted environments. Impacts of non-cooperative and misbehaving nodes, as they are likely to occur in publically accessible deployments on the Internet, have yet been out of our focus and need further investigation.
Another interesting aspect for future research is the definition of load metrics other than the classic element count. We presume metrics taking access frequency into account to be more suitable for real life deployments, since they are able to express load more fine-grained, which is required to coop with massive popularity imbalances e.g. so-called *flash-crowd* effects.

Bibliography

[1] Sonesh Surana et al. “Load balancing in dynamic structured peer-to-peer systems”. In: *Performance Evaluation* 63 (3 2006), pp. 217–240.

[2] Ion Stoica et al. “Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications”. In: *Proceedings of the 2001 ACM SIGCOMM Conference.* 2001, pp. 149–160.

[3] Antony Rowstron and Peter Druschel. “Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems”. In: *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware).* Heidelberg, Germany, Nov. 2001, pp. 329–350.

[4] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. *Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing.* Tech. rep. UCB/CSD-01-1141. EECS Department, University of California, Berkeley, 2001.

[5] Nicholas J. A. Harvey et al. “SkipNet: A Scalable Overlay Network with Practical Locality Properties.” In: *USENIX Symposium on Internet Technologies and Systems.* Seattle, WA, Mar. 2003.

[6] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. *Chord$^{\#}$: Structured Overlay Network for Non-Uniform Load Distribution.* Tech. rep. Konrad-Zuse-Zentrum für Informationstechnik Berlin, Aug. 2005.

[7] Prasanna Ganesan, Mayank Bawa, and Hector Garcia-Molina. “Online Balancing of Range-Partitioned Data with Applications to Peer-to-Peer Systems”. In: *In VLDB.* 2004, pp. 444–455.

[8] M. Ripeanu. “Peer-to-Peer Architecture Case Study: Gnutella Network”. In: *IEEE International Conference on Peer-to-Peer Computing.* Linkoping, Sweden: IEEE Computer Society, 2001, pp. 99–100.

[9] Jian Liang, Rakesh Kumar, and Keith W. Ross. “The FastTrack overlay: a measurement study”. In: *Computer Networks* 50.6 (2006), pp. 842–858.

[10] S. Bakhtiari, R. Safavi-naini, and J. Pieprzyk. “Cryptographic Hash Functions: A Survey”. In: *USENIX Technical Conference.* 1995.

[11] D. Eastlake and P. Jones. *US Secure Hash Algorithm 1 (SHA1).* Tech. rep. 3174. Sept. 2001.

[12] S. A. Baset and H. G. Schulzrinne. “An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol”. In: *Proceedings of the 25th IEEE International Conference on Computer Communications.* INFOCOM 2006. 2006, pp. 1–11.

[13] Salman Abdul Baset, Gaurav Gupta, and Henning Schulzrinne. “OpenVoIP: An Open Peer-to-Peer VoIP and IM System”. In: *In: Proceedings of SIGCOMM’08 (demo).* 2008.

[14] Jonathan Rosenberg et al. *SIP: Session Initiation Protocol.* RFC 3261. Fremont, CA, USA: RFC Editor, June 2002.

[15] Jens Fiedler et al. “Reliable VoIP Services Using a Peer-to-Peer Intranet”. In: *Proceedings of the Eighth IEEE International Symposium on Multimedia.* ISM ’06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 121–130.

[16] Giuseppe DeCandia et al. “Dynamo: amazon’s highly available key-value store”. In: *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles.* SOSP ’07. Stevenson, Washington, USA: ACM, 2007, pp. 205–220.

[17] Avinash Lakshman and Prashant Malik. “Cassandra: structured storage system on a P2P network”. In: *PODC ’09: Proceedings of the 28th ACM symposium on Principles of distributed computing.* New York, NY, USA: ACM, 2009, pp. 5–5.

[18] E. Hewitt. *Cassandra: The Definitive Guide.* Definitive Guide Series. O’Reilly Media, 2010, p. 27.

[19] Petar Maymounkov and David Mazières. “Kademlia: A Peer-to-Peer Information System Based on the XOR Metric”. In: *Revised Papers from the First International Workshop on Peer-to-Peer Systems.* IPTPS ’01. London, UK: Springer-Verlag, 2002, pp. 53–65.

[20] Scott A. Crosby and Dan S. Wallach. *An Analysis of BitTorrent’s Two Kademlia-Based DHTs.* Tech. rep. TR-07-04. Department of Computer Science, Rice University, June 2007.

[21] Zhaosheng Zhu et al. “Botnet Research Survey”. In: *International Computer Software and Applications Conference.* 2008, pp. 967–972.

[22] Yong Liu, Yang Guo, and Chao Liang. “A survey on peer-to-peer video streaming systems”. In: *Peer-to-Peer Networking and Applications* 1 (1 2008), pp. 18–28.

[23] Miguel Castro et al. “SplitStream: high-bandwidth multicast in cooperative environments”. In: *SIGOPS Oper. Syst. Rev.* 37 (5 2003), pp. 298–313.

[24] Dejan Kostić et al. "Bullet: high bandwidth data dissemination using an overlay mesh". In: *Proceedings of the nineteenth ACM symposium on Operating systems principles.* SOSP '03. New York, NY, USA: ACM, 2003, pp. 282–297.

[25] P. Mockapetris and K. J. Dunlap. "Development of the domain name system". In: *Symposium proceedings on Communications architectures and protocols.* SIGCOMM '88. New York, NY, USA: ACM, 1988, pp. 123–133.

[26] Russ Cox, Athicha Muthitacharoen, and Robert Morris. "Serving DNS Using a Peer-to-Peer Lookup Service". In: *Revised Papers from the First International Workshop on Peer-to-Peer Systems.* IPTPS '01. London, UK: Springer-Verlag, 2002, pp. 155–165.

[27] Frank Dabek et al. "Designing a DHT for low latency and high throughput". In: *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1.* San Francisco, California: USENIX Association, 2004, pp. 85–98.

[28] Venugopalan Ramasubramanian and Emin G. Sirer. "The design and implementation of a next generation name service for the internet". In: *SIGCOMM Comput. Commun. Rev.* 34 (Aug. 2004), pp. 331–342.

[29] Venugopalan Ramasubramanian and Emin Gün Sirer. "Beehive: O(1)lookup performance for power-law query distributions in peer-to-peer overlays". In: *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1.* Berkeley, CA, USA: USENIX Association, 2004, pp. 8–8.

[30] Vasileios Pappas et al. "A Comparative Study of the DNS Design with DHT-Based Alternatives". In: *25th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies.* Barcelona, Spain, 2006, pp. 1–13.

[31] Kalman Graffi et al. "LifeSocial.KOM: A P2P-Based Platform for Secure Online Social Networks". In: *Peer-to-Peer Computing.* 2010, pp. 1–2.

[32] Sonja Buchegger et al. "PeerSoN: P2P social networking: early experiences and insights". In: *SNS '09: Proceedings of the Second ACM EuroSys Workshop on Social Network Systems.* New York, NY, USA: ACM, 2009, pp. 46–52.

[33] Sean Christopher Rhea. "Opendht: a public dht service". AAI3211499. PhD thesis. Berkeley, CA, USA, 2005.

[34] Leucio Cutillo, Refik Molva, and Thorsten Strufe. "Safebook: A privacy-preserving online social network leveraging on real-life trust". In: *IEEE Communications Magazine* 47.12 (Dec. 2009), pp. 94–101.

[35] Ozgur D. Sahin et al. “PRoBe: Multi-dimensional Range Queries in P2P Networks”. In: *6th International Conference on Web Information Systems Engineering.* WISE’05. 2005, pp. 332–346.

[36] Sylvia Ratnasamy et al. “A scalable content-addressable network”. In: *SIGCOMM Comput. Commun. Rev.* 31.4 (Aug. 2001), pp. 161–172.

[37] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. “A Structured Overlay for Multi-dimensional Range Queries.” In: *Euro-Par.* Ed. by Anne-Marie Kermarrec, Luc Bougé, and Thierry Priol. Vol. 4641. Lecture Notes in Computer Science. Springer, 2007, pp. 503–513.

[38] C. Schmidt and M. Parashar. “Flexible Information Discovery in Decentralized Distributed Systems”. In: *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC’03).* Washington, DC, USA: IEEE Computer Society, 2003.

[39] Keong Lua et al. “A survey and comparison of peer-to-peer overlay network schemes”. In: *Communications Surveys & Tutorials, IEEE* (2005), pp. 72–93.

[40] Daniel R. Karrels, Gilbert L. Peterson, and Barry E. Mullins. “Structured P2P technologies for distributed command and control.” In: *Peer-to-Peer Networking and Applications* 2.4 (Dec. 11, 2009), pp. 311–333.

[41] Alan Demers et al. “Epidemic algorithms for replicated database maintenance”. In: *SIGOPS Oper. Syst. Rev.* 22 (1 1988), pp. 8–32.

[42] Robbert van Renesse et al. “Efficient reconciliation and flow control for anti-entropy protocols”. In: *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware.* LADIS ’08. New York, NY, USA: ACM, 2008, 6:1–6:7.

[43] Moritz Steiner, Taoufik En-Najjary, and Ernst W. Biersack. “A global view of kad”. In: *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement.* IMC ’07. New York, NY, USA: ACM, 2007, pp. 117–122.

[44] Rusty Klophaus. “Riak Core: building distributed applications without shared state”. In: *ACM SIGPLAN Commercial Users of Functional Programming.* CUFP ’10. New York, NY, USA: ACM, 2010, 14:1–14:1.

[45] K. Gummadi et al. “The impact of DHT routing geometry on resilience and proximity”. In: *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications.* SIGCOMM ’03. New York, NY, USA: ACM, 2003, pp. 381–394.

[46] Dahlia Malkhi, Moni Naor, and David Ratajczak. “Viceroy: a scalable and dynamic emulation of the butterfly”. In: *Proceedings of the twenty-first annual symposium on Principles of distributed computing.* PODC ’02. New York, NY, USA: ACM, 2002, pp. 183–192.

[47] Frans M. Kaashoek and David R. Karger. *Koorde: A Simple Degree-Optimal Distributed Hash Table.* Springer, 2003, pp. 98–107.

[48] H.J. Siegel. “Interconnection Networks for SIMD Machines”. In: *Computer* 12 (1979), pp. 57–65.

[49] N. G. de Bruijn. “A Combinatorial Problem”. In: *Koninklijke Nederlandsche Akademie Van Wetenschappen* 49.6 (June 1946). Ed. by W. Van Der Woude, pp. 758–764.

[50] Mario Schlosser et al. “HyperCuP: hypercubes, ontologies, and efficient search on peer-to-peer networks”. In: *Proceedings of the 1st international conference on Agents and peer-to-peer computing.* AP2PC’02. Bologna, Italy: Springer-Verlag, 2003, pp. 112–124.

[51] Karl Aberer et al. “Improving Data Access in P2P Systems”. In: *IEEE Internet Computing* 6 (1 2002), pp. 58–67.

[52] C. Greg Plaxton, Rajmohan Rajaraman, and Andréa W. Richa. “Accessing nearby copies of replicated objects in a distributed environment”. In: *ACM Symposium on Parallel Algorithms and Architectures.* 1997, pp. 311–320.

[53] William Pugh. “Skip lists: a probabilistic alternative to balanced trees”. In: *Commun. ACM* 33 (6 1990), pp. 668–676.

[54] James Aspnes and Gauri Shah. “Skip graphs”. In: *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms.* SODA ’03. Baltimore, Maryland: Society for Industrial and Applied Mathematics, 2003, pp. 384–393.

[55] Sylvia Ratnasamy, Ion Stoica, and Scott Shenker. “Routing Algorithms for DHTs: Some Open Questions”. In: *Revised Papers from the First International Workshop on Peer-to-Peer Systems.* IPTPS ’01. London, UK: Springer-Verlag, 2002, pp. 45–52.

[56] Michael R. Genesereth and Nils J. Nilsson. *Logical foundations of artificial intelligence.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1987.

[57] Min Cai and Martin Frank. “RDFPeers: A Scalable Distributed RDF Repository based on A Structured Peer-to-Peer Network”. In: *Proceedings of the 13th International World Wide Web Conference (WWW2004).* May 2004, pp. 650–657.

[58] Dominic Battré and Felix Heine and André Höing and Odej Kao. “On Triple Dissemination, Forward-Chaining, and Load Balancing in DHT based RDF stores”. In: *Proceedings of Fourth International Workshop on Database, Information Systems and Peer-to-Peer Computing (DBISP2P 2006).* 2006.

[59] Ananth Rao et al. “Load Balancing in Structured P2P Systems.” In: *IPTPS.* Ed. by M. Frans Kaashoek and Ion Stoica. Vol. 2735. Lecture Notes in Computer Science. Springer, 2003, pp. 68–79.

[60] Jonathan Ledlie and Margo I. Seltzer. “Distributed, secure load balancing with skew, heterogeneity and churn.” In: *INFOCOM.* IEEE, 2005, pp. 1419–1430.

[61] John Byers, Jeffrey Considine, and Michael Mitzenmacher. “Simple Load Balancing for Distributed Hash Tables”. In: *2nd International Workshop on Peer-to-Peer Systems (IPTPS 03).* Springer, 2003, pp. 31–35.

[62] David R. Karger and Matthias Ruhl. “Simple Efficient Load Balancing Algorithms for Peer-to-Peer Systems”. In: *SPAA ’04: Proceedings of the sixteenth annual ACM Symposium on Parallelism in Algorithms and Architectures.* New York, NY, USA: ACM Press, 2004, pp. 36–43.

[63] Yingwu Zhu and Yiming Hu. “Efficient, Proximity-Aware Load Balancing for DHT-Based P2P Systems”. In: *IEEE Transactions on Parallel and Distributed Systems* 16.4 (2005), pp. 349–361.

[64] Manolis Koubarakis et al. “Semantic Grid Resource Discovery using DHTs in Atlas”. In: *3rd GGF Semantic Grid Workshop.* Feb. 2006.

[65] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. “LUBM: A Benchmark for OWL Knowledge Base Systems”. In: *Journal of Web Semantics* 3.2 (2005), pp. 158–182.

[66] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. “Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema”. In: *ISWC ’02: Proceedings of the First International Semantic Web Conference on The Semantic Web.* London, UK: Springer-Verlag, 2002, pp. 54–68.

[67] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. “Range queries on structured overlay networks”. In: *Computer Communications* 31 (2 2008), pp. 280–291.

[68] Miguel Castro et al. “Future directions in distributed computing”. In: Berlin, Heidelberg: Springer-Verlag, 2003. Chap. Topology-aware routing in structured peer-to-peer overlay networks, pp. 103–107.

[69] Dragan Milic and Torsten Braun. “Fisheye: Topology aware choice of peers for overlay networks”. In: *IEEE Conference on Local Computer Networks.* IEEE, 2009, pp. 467–474.

[70] Weiyu Wu et al. "LDHT: Locality-aware Distributed Hash Tables". In: *International Conference on Information Networking. ICOIN 2008.* 2008, pp. 1 –5.

[71] Martin Raack et al. "Papnet: A Proximity-aware Alphanumeric Overlay Supporting Ganesan On-Line Load Balancing". In: *ICPADS '09: Proceedings of the 2009 15th International Conference on Parallel and Distributed Systems.* Washington, DC, USA: IEEE Computer Society, 2009, pp. 440–447.

[72] Krishna P. Gummadi, Stefan Saroiu, and Steven D. Gribble. "King: Estimating Latency between Arbitrary Internet End Hosts". In: *SIGCOMM Internet Measurement Workshop 2002.* 2002.

[73] Antonin Guttman. "R-trees: A Dynamic Index Structure for Spatial Searching". In: *International Conference on Management of Data.* ACM, 1984, pp. 47–57.

[74] Verena Kantere, Spiros Skiadopoulos, and Timos Sellis. "Storing and Indexing Spatial Data in P2P Systems". In: *IEEE Trans. on Knowl. and Data Eng.* 21 (2 2009), pp. 287–300.

[75] Anirban Mondal and Yilifu Masaru Kitsuregawa. "P2PR-tree: An R-tree-based spatial index for peer-to-peer environments". In: *In Proceedings of the International Workshop on Peer-to-Peer Computing and Databases (held in conjunction with EDBT.* Springer-Verlag, 2004, pp. 516–525.

[76] Egemen Tanin, Aaron Harwood, and Hanan Samet. "Using a distributed quadtree index in peer-to-peer networks". In: *VLDB Journal* 16 (2007), pp. 165–178.

[77] Prasanna Ganesan, Beverly Yang, and Hector Garcia-Molina. "One torus to rule them all: multi-dimensional queries in P2P systems". In: *Proceedings of the 7th International Workshop on the Web and Databases: colocated with ACM SIGMOD/PODS 2004.* WebDB '04. Paris, France, 2004, pp. 19–24.

[78] Bongki Moon et al. "Analysis of the Clustering Properties of the Hilbert Space-Filling Curve". In: *IEEE Transactions on Knowledge and Data Engineering* 13 (2001), pp. 124–141.

[79] Morton. *A computer oriented geodetic data base and a new technique in file sequencing.* Tech. rep. Ottawa, Ontario, Canada. 1966.

[80] T. H. Cormen et al. *Introduction to Algorithms.* New York: The MIT Press, 2001.

[81] D. T. Lee and C. K. Wong. "Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees". In: *Acta Informatica* 9.1 (Mar. 1977), pp. 23–29.

[82] Dominic Battre et al. “Extending Pastry by an Alphanumerical Overlay”. In: *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*. CCGRID ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 36–43.