Department of Electronic and Computer Engineering
Technical University of Crete

This is for the obtainment of the Diploma of Electronic and Computer Engineer at the Technical University of Crete

P2P Multidimensional Range Queries
(GRaSP: Generalized Range Search in P2P Networks)

Michail ARGYRIOU

The committee is consisted of:

Vasilis SAMOLADAS (supervisor)
Euripides G.M. PETRAKIS
Stavros CHRISTODOULAKIS

August 28, 2008

Abstract

In this paper we will present the Generalized Range Search over P-Grid (GRaSP) framework. GRaSP provides a model and an API for constructing novel distributed data structures than can handle generalized range queries. This means that we can customize GRaSP on the shape and dimensionality of the data and queries and will produce source code for a distributed data structure that can handle the predefined type of queries. We exhibit and evaluate empirically GRaSP by implementing two protocols on in. First of all the Multidimensional Range Search protocol (MDRS) and secondly the Three-sided Range Search protocol (3SIDED). MDRS can answer d-dimensional rectangular range queries and 3SIDED can answer d-dimensional 3-sided range queries. 3SIDED is in our knowledge the only distributed data structure that can handle 3-sided range queries. Experiments verify the theoretically logarithmic to the number of the peers on the network latency and the low maximum throughput, i.e. the load of the most loaded peer on the network.

List of Figures

List of Algorithms

Contents

Chapter 1

Introduction

Searching is a fundamental problem in Computer Science for decades now. The problem is to organize pairs of ordered keys and values so that the retrieval of keys (and consequentially their values) given a range is efficient. This problem is called the *Generalized Range Search Problem*. More formally the Generalized Range Search is defined as following. Assume we denote the search space with the symbol U (infinite set), the key space with $\mathcal{K} \subseteq 2^U$ (keys are subsets of U) and the range space with $\mathcal{R} \subseteq 2^U$ (ranges are subsets of U). Then assuming that the dataset $K \subset \mathcal{K}$ is stored on a network and given a general range query $r \subseteq R$ we want all the keys $A_K(r) = \{k \in K | k \cap r \neq \emptyset\}$ that answer it. Note that we don't assume beforehand the shape nor the dimensionality of the data or the queries. This means that the data can be points, rectangles[1], polygons or whatever. Also note that nowhere do we mention the nature of the storage medium, that is if the keys are stored on the Primary Memory of the Secondary Memory of on Peer-2-Peer (P2P) network.

The Generalized Range Search problem can be instantiated in the form of *Point Search Problem* where the keys and the queries are points. The answer to a query are all the keys that are identified with the query. Another problem is the *Orthogonal Range Search Problem* where the keys can be of any shape and the query is a rectangle. The answers to a query are all the keys that overlap the query.

Software packages that can handle such generalized range search problems are called *Frameworks*[2]. A framework is customized for a specific type of search problem (i.e. shape and dimensionality of data and queries) and generates a "solution" data structure. Nowadays such frameworks for Primary and Secondary Memory exist with satisfactory performance. On the other hand rudimentary work has been done for the case of P2P. Therefore the need is imperative for developing frameworks that produce efficient distributed data structures. Our work comes to complete this gap. We have crafted a framework called *Generalized Range Search over P-Grid* (*GRaSP*) which tackles the aforementioned generalized range search problem more efficiently than any existing adversary.

Initially we will present the most prominent data structures that tackle individual search problems such as point and rectangles range queries for (in this order) Primary Memory, Secondary Memory and P2P. On the next chapter we will present existing framework solutions.

For the Primary Memory the best well-known data structures are the Binary Search Tree[36] with worst-case search cost $O(\log n)$, balanced data structures such as 2-3-4 tree[36], AVL[36] and

[1] We interchangeably use the terms *rectangle*, *rectangular* and *orthogonal*.

[2] We interchangeably use the terms *framework*, *protocol* and *network*.

Red-Black Tree[36] with worst-case search cost $O(\log n)$ but with slight increased construction and update costs in order they remain balanced, and data structures such the Splay Tree[36] which have expected search cost $O(\log n)$. On Secondary Memory the balanced solutions dominate with the most well-known cases of *2-3-4 tree*[36], *B-tree*[14], $B^+ - tree$ and $B^* - tree$ with worst-case search cost $O(\log_B n)$, where B is the number of keys that can fit in one block and n is the number of the keys. *UB-tree*[15] is an interesting extension of the $B^+ - tree$ for handling multi-dimensional keys.

Having solved the point search problem the interest nowadays lies on rectangle range search. Orthogonal range search comes in many variants. The most simple one is the 1-dimensional Orthogonal range search where we want to retrieve all keys within a query interval.A well-known data structure for answering 1-dimensional range queries is the *Interval Tree*[22] (or *Segment Tree*) which stores 1-dimensional intervals and returns all the ranges overlapping the query. Query needs $O(\log n + k)$ time, construction of the tree $O(n \log n)$ time and storage $O(n)$ space, where n is the number of stored objects and k is the number of reported results. The Interval Tree has been externalized (*External Interval Tree*) and needs $O(n/B)$ space, $O(\log_B n + T/B)$ I/Os for searching and $O(\log_B T)$ I/Os for updating, where B is the number of objects per disk block and T is the number of reported results.

Generalizing the dimensionality of the 2-dimensional Orthogonal range search problem into higher dimensions a multi-dimensional Orthogonal range search retrieves all the keys within a query hyper-rectangle. The generalization of the Interval Tree for higher dimensions is the *Range Tree*[36] which needs for searching $O(\log^2 n + k)$ time and for storage $O(n \log n)$ space. *External Range Tree* is the external extension of the Range tree and needs $O(\log_B n + T/B)$ I/Os for searching and $O(\log_B^2 n / \log_B \log_B n)$ I/Os for insertions/deletions. Another data structure that can handle multi-dimensional (k-dimensions) Orthogonal range queries is the well-known *kd-tree*[17]. Levels of the tree are split along successive dimensions at the points. kd-tree can also be used for nearest neighbor searches. The complexity (if the tree is balanced) is $O(n \log^2 n)$ for the construction of the tree, $O(\log n)$ for insertion, $O(\log n)$ for removal and $O(n^{1-1/d} + k)$ for searching. An interesting extension of the kd-tree is the *adaptive k-d tree*[18] where successive levels may be split along different dimensions. An interesting extension of it, especially useful for the disk, is the *k-d-B-tree*[39] which is actually a kd-tree in the sense that it splits multidimensional spaces like an adaptive k-d tree, but also balances the resulting tree like a B-tree. k-d-B-tree is a static tree because it doesn't allow point insertions nor deletions. It needs $O(\sqrt{n/B} + T/B)$ I/Os for searching and $O(n/B \log_B n)$ I/Os for construction. If we make it dynamic using a logarithmic method then the searching needs $O(\sqrt{n/B} + T/B)$ I/Os, the updates $O(\log_B^2 n)$ I/Os and $O(n/B)$ space (linear space). Probably the most general and applicable data structure is the *R tree*[28]. R tree stores multidimensional objects (eg intervals, regions, 3-D objects, arbitrary high dimensional objects). Actually it is the generalization of the B-tree[14] for higher dimensions and for storing more general objects. Many extensions of the R tree have been developed but just to mention a few ones there is the $R^+ - tree$, the $R^* - tree$ and the Hilbert R-tree. $R^+ - tree$[40] is used for indexing spatial information and is a compromise between the R-tree and the K-D-B tree; it avoids overlapping of internal nodes by inserting an object into multiple leaves if necessary. $R^* - tree$[16] supports point and spatial data at the same time with a slightly higher cost than other R-trees. *Hilbert R-tree*[34] exploits the fact that the performance of the R-trees depends on the quality of the algorithm that clusters the data rectangles on a node. More precisely Hilbert R-tree uses Hilbert curve to impose a linear ordering on the data rectangles. It can be thought of as an extension of $B^+ - tree$ for multi-dimensional objects. Another data structure for multi-dimensional Orthogonal range search is the *Quad tree*[26] which stores areas, points, lines and curves. It can be easily extended into the 3-dimensional space (*Octal Tree*[20]). *O-tree*[41] is used

for planar planar Orthogonal range search on external memory. It needs $O(\sqrt{n/B} + T/B)$ I/Os for searching, $O(\log_B n)$ I/Os for insertions/deletions and $O(n/B)$ space (linear space). It can be extended to work in d-dimensions with optimal query bound $O((n/B)^{1-1/d} + T/B)$.

Let's consider now P2P data structures. The trend nowadays is to map data and peers into IDs. One way to achieve this is via the adoption of the DHTs. A *Distributed Hash Table* (*DHT*) is actually a distributed Hash Table. The Hash Table is an efficient data structure (for Internal and External Memory) for storing pairs of {key,value} and retrieving in $O(1)$ (expected) time a value if you have the key. Similarilly, a *Distributed Hash Table* (*DHT*[4, 2, 3, 5]) is a way to look up (usually in $O(logn)$) a value into a structured, decentralized, scalable and fault-tolerant network if you have the key (n is the number of the peers). This implies that in a network we deterministically store keys (instead of values) to peers and probably on foreign peers instead on the owner peer of each value. In order to find the responsible peer for a value, we hash the value into a key ID, we also hash (using the same hash function) to a unique peer credential (such as the peer IP) to a peer ID and afterwards we store (or retrieve) the key to (from) peer which has the most similar ID to the key ID (or alternatively "the less distant"). The distance metric among key IDs and peer IDs is defined each time by the protocol employed; for example the distance between a key and a peer ID is defined by Kademlia[5] as their XOR distance whereas Chord[2] defines it as their difference; note that XOR is a symmetrical distance metric versus the difference. In order to look up for an object we retrieve its key (by hashing it) and in a step-by-step process a relative query is forwarded through the overlay until it reaches the peers that hold the key. Some of the networks that use a DHT are the CAN[4], MURK[8], Kademlia[5], Pastry[3], Tapestry[6], Chord[2] and PGrid[10].

Point searching is the oldest and best studied searching problem on P2P and many networks have been developed such as Pastry [3], Kademlia [5], Chord [2] and PGrid [10] which answer a query in $O(logn)$, where n is the number of the peers. Moreover, SkipGraphs [13] and SkipNet[29] can handle point queries with cost $O(logn)$ w.h.p. and congestion $O(logn)$. *Rainbow Skip Graphs* [27] provide essentially the same performance guarantees, but with additional guarantees for fault tolerance. SkipWebs [12] can handle multi-dimensional point queries with expected cost $O(logn/loglogn)$ (same for updates) and $O(logn)$ congestion. Other methods that can tackle d-dimensional point queries are CAN [4] and MURK [8].

An easy way to make DHTs able to handle multi-dimensional queries is mapping the d-dimensional keys onto 1-D with the help of a hashing algorithm (such as space-filling curves). Ganesan et al. [8] propose two structures, SCRAP, based on space-filling curves over a skip graph-like network, and MURK, a CAN-derived network which partitions space in a manner similar to k-d trees. They evaluate their techniques experimentally, but do not consider congestion. Moreover the *shower algorithm* [24] facilitates 1-dimensional range queries over a trie by using space filling curve.

BATON[31] is a distributed balanced tree based method which can handle 1-dimensional Orthogonal range queries in $O(logn)$ time and updates in $O(logn)$ amortized time, where n is the number of peers. $BATON^*$ is a k-ary tree generalization of BATON with query cost $O(log_k n)$ and update cost $O(k + logn)$. Another protocol is the P-tree [23].

To go one step further PHT [38], MURK [8] and Distributed Segment Tree [44] can support multi-dimensional Orthogonal range search.

Abstracting the shape and dimensionality of a range query we now face a more general search problem, i.e. generalized range search . A generalized range search problem is formally defined

by a pair $(\mathcal{K}, \mathcal{R})$, where $\mathcal{K}$ is the set of keys, and $\mathcal{R} \subseteq 2^{\mathcal{K}}$ is the set of the query ranges. Our ultimate goal is to organize a finite set $K \subseteq \mathcal{K}$ so that, for any $r \in \mathcal{R}$, the $r \cap K$ can be computed efficiently. Or simpler stated we want to retrieve all the keys that answer a range query as fast as possible (in terms of either memory accesses or disk accesses or messages).

Having defined the generalized range queries we now want a framework which we can customize to particular search problems. Such a framework should provide a theoretical model and a API to develop new data structures that will have some theoretical efficiency safeguards. The typical procedure of the development of a new data structure using a framework is initially customizing the framework and afterwards generating the source code for the data structure in question. Any physical issues such as memory management, I/O and message passing should be hidden from the user in order to decrease complexity and development time.

On this thesis, we are especially interested on frameworks that can produce distributed data structures, i.e. data structures that can handle queries over a P2P environment. Assume for example that we want to construct a distributed data structure that can answer point queries. In this case, we could address to such a P2P framework, customize it with information such as how the peers organize, the state of each peer and when a peer can answer the query and the framework would generate source code for such a data structure ready to be used. Issues such as the middleware needed for the peers to communicate should be hidden from the user and taken care automatically from the framework. The framework could also be more clever and any organization information be hidden and self-handled by the framework. For example the peers could be organized into a list or over a tree or over a skip-list etc.

We propose a novel P2P framework called *Generalized Range Search over P-Grid* (*GRaSP*) which tackles the generalized range search problem in a distributed manner. One can easily customize GRaSP and rapidly develop new data structures that are overlayed over a network. Theoretical guarantees and boundaries are given for its performance. More specifically on GRaSP range queries are answered on $O(\log n)$ hops with high probability, where n is the number of peers of the network. The congestion is also provable low, i.e. $O(\log n)$ and there is also the ability to introduce storage redundancy to improve load balancing. Lastly, we empirically evaluate GRaSP by implementing two new protocols on it. The first protocol is the Multidimensional Range Search (MDRS) and can deal with the d-dimensional rectangle queries. The second one is the Three-sided Range Search (3SIDED) protocol and can deal with d-dimensional 3-sided queries. In our knowledge 3SIDED is the only network that can deal with 3-sided queries over a network.

On Chapter 2 we introduce the reader to existing frameworks for P2P that tackle the generalized search problem with especially emphasis on the PGrid case. On Chapter 3 we present GRaSP. On the next two chapters we implement two networks based on GRaSP, i.e. on Chapter 4 we present the MDRS network which can answer efficiently d-dimensional Orthogonal range queries and on Chapter 5 we present the 3SIDED network which can answer 3-sided queries. On Chapter 6 we evaluate MDRS and 3SIDED and we conclude on Chapter 7.

Chapter 2

Related Work

In Chapter 1, we emphasized the central role that searching plays in the field of Computer Science. We mentioned that searching is so far well-studied for Primary and Secondary Memory but immature yet for P2P environments. We are especially interested on constructing a framework which will tackle the Generalized Range Search Problem and facilitate the rapid development of any distributed data structure.

The concept of frameworks is very old. For example in the field of Software Engineering several attempts had been made to construct frameworks that allowed the rapid composition and generation of new systems. A typical framework was GENESIS[1] on 1990. Soon the concept of frameworks was developed by the Databases field. Maybe the most prominent example of such a framework is the GiST framework which allows the development of data structures for the Secondary Memory. Extending these ideas we want a framework that can be easily customized and produce distributed data structures. Typical examples of such frameworks are VBI[9] and PGrid[10] networks.

On this chapter we will describe GiST, VBI and PGrid. We will delve more into PGrid because it's peer organization will be the inspiration for GRaSP. At the end of this chapter we to be clear the pros and especially the cons of VBI and PGrid that will introduce on next chapter the superior GRaSP.

2.1 GiST

GiST[30] is a tree data structure which supports search and update functions and provides an API which further supports recovery and transactions. GiST framework can be used to build a variety of search trees for Secondary Memory such as R-Tree[28], B-Tree[14], hB-tree[35] and RD-tree[43]. Not surprisingly GiST has been used to construct many indices for the well-known ORDBMS PostgreSQL[1].

[1] http://www.postgresql.org

2.2 VBI

A distributed data structure oriented to generalized range queries is the *Virtual Binary Index Tree* (*VBI-tree*)[9]. On VBI peers are overlayed[2] over a balanced binary tree like they do on BATON[31]. The tree is only virtual, in the sense that peer nodes are not physically organized in a tree structure at all. The abstract methods defined can support any kind of hierarchical tree indexing structures in which the region managed by a node covers all regions managed by its children. Popular multidimensional hierarchical indexing structures that can be built on top of VBI include the R-tree[28], the X-tree[19], the SS-tree[25], the M-tree[21], and their variants. VBI guarantees that point queries and range queries can be answered within $O(logn)$ hops, where n is the number of the peers. VBI specifies an effective load balancing strategy to allow nodes to balance their work load efficient. Validation has been made by applying the M-tree and nearest neighbor queries over VBI.

The major drawback of VBI is that it isn't scalable for low-dimensional rectangular range queries as has been recently proved by Blanas et al[42]. They compared VBI[9], PGrid[10], CAN[4] and MURK[8] and concluded that the only scalable network is PGrid.

2.3 PGrid

According to the results of Blanas et al[42] (see the previous paragraph) the most scalable network if PGrid[10]. Our framework GRaSP borrows many elements from PGrid and therefore we delve into PGrid in great detail. Some concepts initially introduced on the following paragraphs are repeated or referred later on on Chapter 3 when speaking for GRaSP.

2.3.1 Topology

The *topology* of the network is referred to the overlay organization of the peers. In PGrid peers are organized over a binary tree but on the contrary to VBI the tree is a trie[36][3] and not necessarily balanced. The trie contains two kinds of nodes; the internals and the externals (leaves). Peers are located only on the leaves and therefore there is no congestion near the root. The internal nodes are virtual in the sense that they do not contain any keys, nor are used and nor are adopted by the peers. Every node has zero or two children. Therefore the trie is characterized as a *virtual trie*. Each peer holds only a part of the overall tree. Each peer is labeled with a unique bitstring (i.e. it contains only 0's and 1's) which is called *PeerID*. PeerID is determined by the peer's position onto the trie and represents the part of the tree that the peer is responsible for. The root has the special PeerID ϵ which denotes the empty bitstring and the overall data space. If a node has $PeerID = pid$ and has children then its children have PeerIDs $pid \cdot 0$ and $pid \cdot 1$, where $\cdot$ denotes the concatenation of two strings. PGrid doesn't relate the PeerID of a peer with its data space.

[2] An *overlay network* is a computer network which is built on top of another network. Nodes in the overlay can be thought of as being connected by virtual or logical links, each of which corresponds to a path, perhaps through many physical links, in the underlying network. For example, many peer-to-peer networks are overlay networks because they run on top of the Internet. (definition from http://en.wikipedia.org/wiki/Overlay_network)

[3] A trie[36] T for a string S belonging into an alphabet Σ is an ordered tree with the following properties: (a) each node (except for the root) is labeled with a character of Σ, (b) the ordering of the children of an internal node of T is determined by a canonical ordering of the alphabet Σ, and (c) T has s external nodes, each associated with a string of S, such that the concatenation of the label of the nodes on the path from the root to an external node v of T yields the string of S associated with v.

Therefore is characterized as a framework.

PGrid construction is triggered by local interactions only. Whenever two peers meet the refine their routing tables with the help of the *exchange algorithm*. More specifically when two peers meet at random or intentionally (for example due to a point search or a datum update) they divide the search space and each one takes responsibility for one half and stores the address of the other peer to cover the other half. Therefore a peer can guarantee the routing of a message to any peer of the trie. There is also an algorithm for the construction of the trie if there is already a pool of peers available. This algorithm gives a balanced trie but the trie may soon become unbalanced because of the exchange algorithm.

An exemplary trie is depicted on Figure2.1.

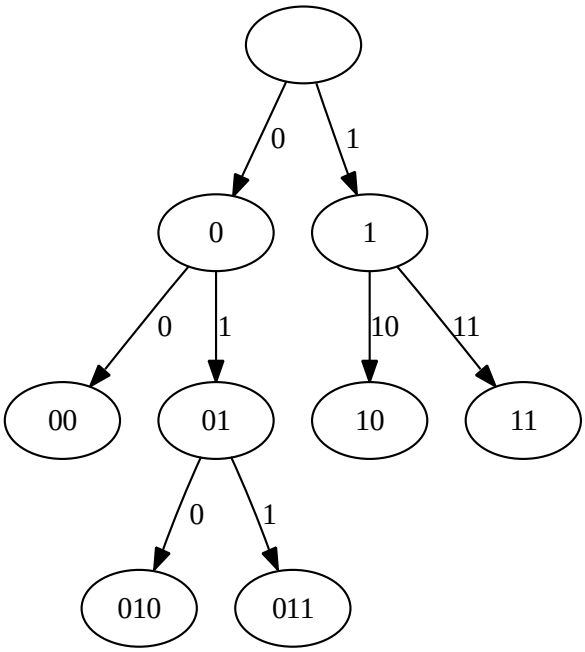


Figure 2.1: PGrid exemplary trie. Each node is labelled by each depth from the root.

2.3.2 Routing Tables

Each peer stores information necessary for routing a message to any peer on the network. The information is in the form of pointers to other peers and is stored on a table, called *Routing Table*. On PGrid, the routing table of each peer is consisted of at least one pointer for each bit of its PeerID to at least another peer with this PeerID as a prefix. This guarantees that if a peer cannot answer a query it can forward it to another peer that is closer to the result.
More specifically, for each bit in its path, a peer stores in its routing table the address of at least one other peer that is responsible for the other side of the binary tree at that level. Thus, if a peer receives a query string it cannot satisfy, it must forward it to a peer that is closer to the result. The PGrid construction algorithm guarantees that peer routing tables always provide at least one path from any peer receiving a request to one of the peers holding a key so that any query can be satisfied regardless of the peer queried. The routing tables are updated through the exchange algorithm which we mentioned in the previous paragraph.

2.3.3 Searching

PGrid was initially developed for answering 1-dimensional point queries[11]. Later on Aberer[24] proposed a searching algorithm which can answer d-dimensional range queries. More specifically using a space filling curve we can map a d-dimensional space into 1-dimensional space. Then we can execute a 1-dimensional range query on the mapped space instead of the original and retrieve there the answers. Aberer's searching algorithm comes into two forms: a sequential algorithm (*Minmax Algorithm*) and a superior parallel (*Shower Algorithm*). In the shower algorithm each peer forwards simultaneously a query it receives to neighbors that can answer it. The cost for the shower algorithm is $O(\log n)$, where n is the number of the peers. The evaluation of the aforementioned searching algorithm has been made theoretically and empirically on PlanetLab using the search algorithms over PGrid.

Here we present the Shower Algorithm. We use our own notation and rename Shower Algorithm into *Route Algorithm*. The purpose for this is that we will we also use the same notation later on when presenting GRaSP on Chapter 3 and its instances MDRS on Chapter 4 and 3SIDED on Chapter 5. Assume that $p \uparrow q$ denotes the Longest Common Prefix of PeerIDs p, q. Also denote with L_p the routing table of peer p. $L_p[x]$ selects uniformly among all q so that $p \uparrow q = x$.

Then apply recursively:

Route(Peer p**, Peer** q**) {**
    if($p \uparrow q = q$ **)**
        Process();
    else
        Route($L_p[p \uparrow q]$ **,** q **);**
}

We explain the Route Algorithm through an example that is depicted on Figure 2.2. Assume that peer 11 wants to route a message to peer 010. Further assume that peer 11 has references to peer 00, peer 00 to 011 and peer 011 to 010. Initially peer 11 forwards a message to peer 00 because 00 has common prefix of length 1 with the target peer 010. Next, peer 00 forwards a message to peer 011 because the latter peer has common prefix of length 2 with the target peer. Likewise does the peer 011 which forwards a message to the target peer. We observe that at each step a message is forwarded to a peer that is at least one bit closer to the target peer. Therefore the routing cost is $O(\log n)$, where n is the number of the peers.

The problem with such space filling curve techniques is that the spatial locality of the data items is not preserved, i.e. if two data items are near on the original space then they are note necessarily near on the reduced mapped space. Therefore neighbor data items can be stored into different peers which are far away onto the underlying PGrid trie.

2.3.4 Updates of Data Items

PGrid supports key updates by utilizing a general algorithm for updates which is basically a hybrid push/pull rumor spreading algorithm which also offers probabilistic guarantees. For a detailed description of the updates see [7].

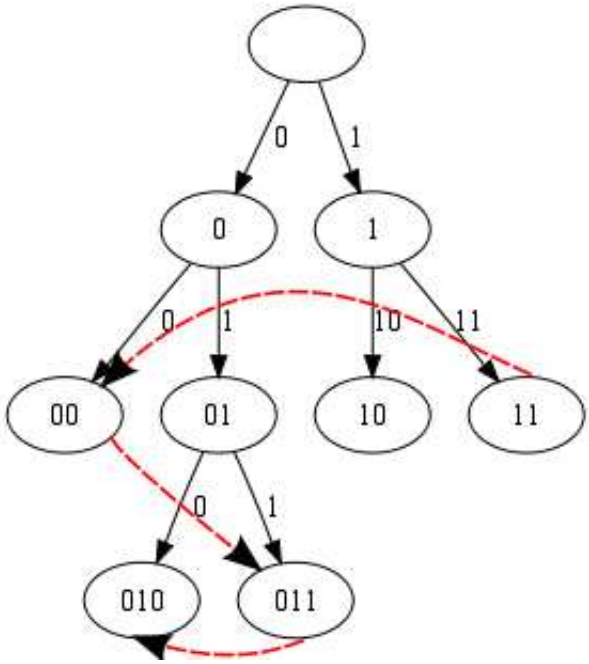


Figure 2.2: Application of the Route Algorithm over the trie of Figure 2.1. Peer 11 wants to forward a message to target peer 010. Initially, peer 11 forwards a message to peer 00 because it has a common prefix of length 1 with the target peer 010. Likewise peer 00 forwards a message to peer 011, and 011 forwards a message to target peer 010. Observe that the distance is halved at each step. The latency here is $H(11, 010) = 3$ hops.

Chapter 3

GRaSP

In the previous chapter, we explored the existing frameworks that tackle the generalized range search problem on P2P. We emphasized on the fact that none of them is a panacea, neither VBI nor PGrid. Therefore we look to a new framework called *Generalized Range Search in P2P Networks (GRaSP)* that also tries to tackle the generalized range search problem on P2P. In general lines GRaSP resembles PGrid on the way the peers are organized over a trie but abstracts the searching algorithm.

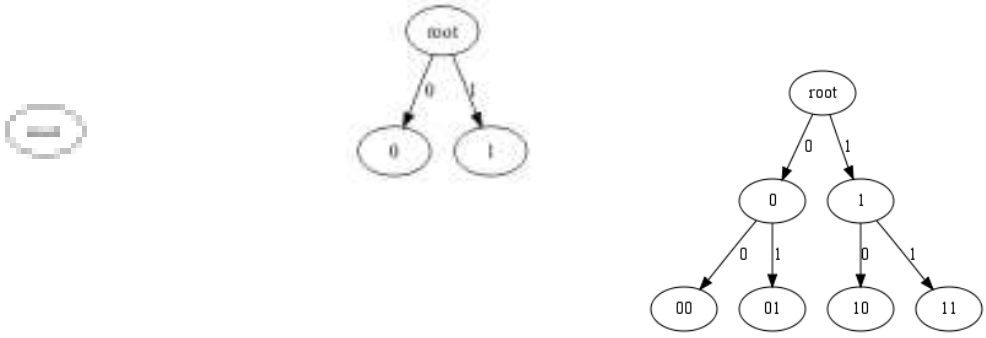
One can customize GRaSP in order to rapidly construct new protocols. The meaning each time will be obvious from the collocation. For example one may want to handle queries of T-type and D-dimensionality. Then he should customize GRaSP for the data space each peer is responsible for, when a peer can answer such a T-type query and how a new peer selects a parent peer. It's obvious that the type of the query and the dimensionality of the dataset should be known beforehand the customization of a new protocol.

The chapter is organized as follows. Initially we present the topology of GRaSP and how the peers organize and share the underlying data space. Afterwards, we study the procedure for data updates (insertions/removals). Next, we present the steps taken by a new peer when joining the network. On next section, we present the state of each peer which allows it to route a message to any peer of the network. This routing mechanism naturally introduces us on the next section with the searching algorithm. Lastly but not least, we sum up the steps needed to be followed in order to customize GRaSP and construct a new protocol. We see that they are straight-forward and easy to be grasped.

3.1 Topology

The peers of the network are organized over a trie samewise as the do on PGrid (see Section 2.3.1). The trie structure has been chosen for the following reasons:

- PGrid (and therefore the underlying trie organization) has been empirically proved scalable to the number of peers.
  Blanas et al[42] have empirically proved that PGrid which is overlayed over a trie) scales better than VBI[9], CAN[4] and MURK[8] in relation to multidimensional orthogonal range queries.
- Aberer [10] has proved that for any trie, the expected hop distance $H(p, q)$ between any pair of peers p,q, is $O(logn)$.



(a) (b) (c)

Figure 3.1: The evolution of the trie while constructing it from a pool of four peers. On Figure 3.1(a) there is only one peer, the root which has $PeerID = \epsilon$. On Figure 3.1(b) a new peer joins the trie. The ex-root becomes node 0 and the new peer becomes node 1. On Figure 3.1(c) two more peers join the trie, peers 0 and 1 become nodes 00 and 10 and the new peers get the PeerIDs 01 and 11 (note that the trie is not necessarily balanced, just happens here).

- More recently Argyriou et al [37] have proven that the routing diameter of any trie with n peers is $O(\log n)$ with high probability.
- Argyriou et al [37] have also proven that the congestion is $O(\log n)$ for any trie with n peers.

For clarification purposes we present on Figure 3.1 the step-by-step evolution of an exemplary trie while new peers join it.

3.2 Hierarchical Space Partitioning

So far we have described how the peers are organized over a trie. Now we will describe which part of the data space each node of the trie adopts. This is referred as the *Space Partitioning* of the key space K. Remember that only the leaves really store keys and not the inner nodes which are virtual. The issue of space partitioning is very important because the load balancing is depended by it. Ideally the data and the queries would be uniformly distributed across the peers. Then all the peers would sustain the same load. But usually this is not the case. Nor the data nor the queries are uniformly distributed. Instead the data distribution is skewed and only a few peers are most popular answering most of the queries. These peers are the bottleneck of the network that limits its scalability. When deciding a space partitioning algorithm for a distributed data structure (i.e. network) one should exploit the nature of the problem. This means that more peers should be located on areas where data are more dense assuming that such areas accept most of the queries. On Chapters 4 and 5 we will provide the reader with two examples of such space partitioning techniques that are provably efficient as is empirically proved on Chapter 6.

Now we pose a few conditions that any hierarchical space partitioning algorithm should meet. If we denote the universal data space by U and the data space of peer $PeerID$ with $S(peerID)$ then we require both the following conditions to comply:

- $S(\epsilon) = U$

- $S(x \cdot 0) \cup S(x \cdot 1) = S(x)$

The first condition states that the root node is responsible for the whole data space. The second one states that the combined data spaces of two sibling nodes equals the data space of their parent peer.

With only these assumptions we note that *a key may be stored into multiple peers*. This is deliberately happening for load balancing purposes and should be controlled by the user's options when defining $S(PeerID)$.

On Figure 3.2 we present the evolution of the trie while new peers join the network and two respective arbitrary space partitioning schemes over a supposed 2-dimensional data space. Note that the trie is the same as the one on Figure 3.1 but augmented with the data space $S()$ of each node. Note that any space partitioning scheme can be chosen at each customization of the GRaSP protocol and therefore beget unique novel protocols (data structures). The first data space partitioning algorithm followed here splits the data space once almost horizontally and once vertically and the second algorithm splits the data space of each peer always horizontally — these are really dummy splitting algorithms and are only used for demonstration purposes. The data space partitioning algorithm of each crafted protocol should ameliorate any congestion problems by exploiting the type of queries, i.e. should locate more peers wherever there are dense data and the opposite on the case the data are terse. We will look a concrete example when studying the 3SIDED protocol on Section 5.1.

3.3 Proposed Bootstrapping Algorithms

In order a new peer to join the network it should first select a peer (*Bootstrap Peer* which will guide its join. The selection algorithm is called *Bootstrapping Algorithm*. For sake of simplicity we assume that the bootstrap peer will adopt the new peer as its child (alternatively it could redirect it to another peer). We now want to propose some algorithms for choosing the new peer a bootstrap peer. For sake of simplicity we propose two bootstrapping algorithms that both assume global knowledge of the peers or that keys. Many more can bootstrapping algorithms can be crafted even free from such limiting assumptions.

Volume Balanced Selection The new peer chooses a random point on the data space (which may not correspond to an existing data key) in the multidimensional search space and the bootstrap node is the node which is responsible for the data space that contains the chosen key.

Data Balanced Selection The new peer chooses a random data key (which exists!) and the bootstrap node is its owner peer.

On the Volume Balanced Selection peers tends to equalize the volume of each peer. On the Data Balanced Selection peers tend to have equal number of keys.

Both algorithms drive to much too much different tries. A delegate exception consists the case where the data are uniformly distributed. Then both selection algorithms drive to similar tries.

3.4 Generalized Searching

Here we present a novel searching algorithm called *Search Algorithm*. This algorithm is an extension of the Route Algorithm already mentioned on Section 2.3. But first let's remind the reader the

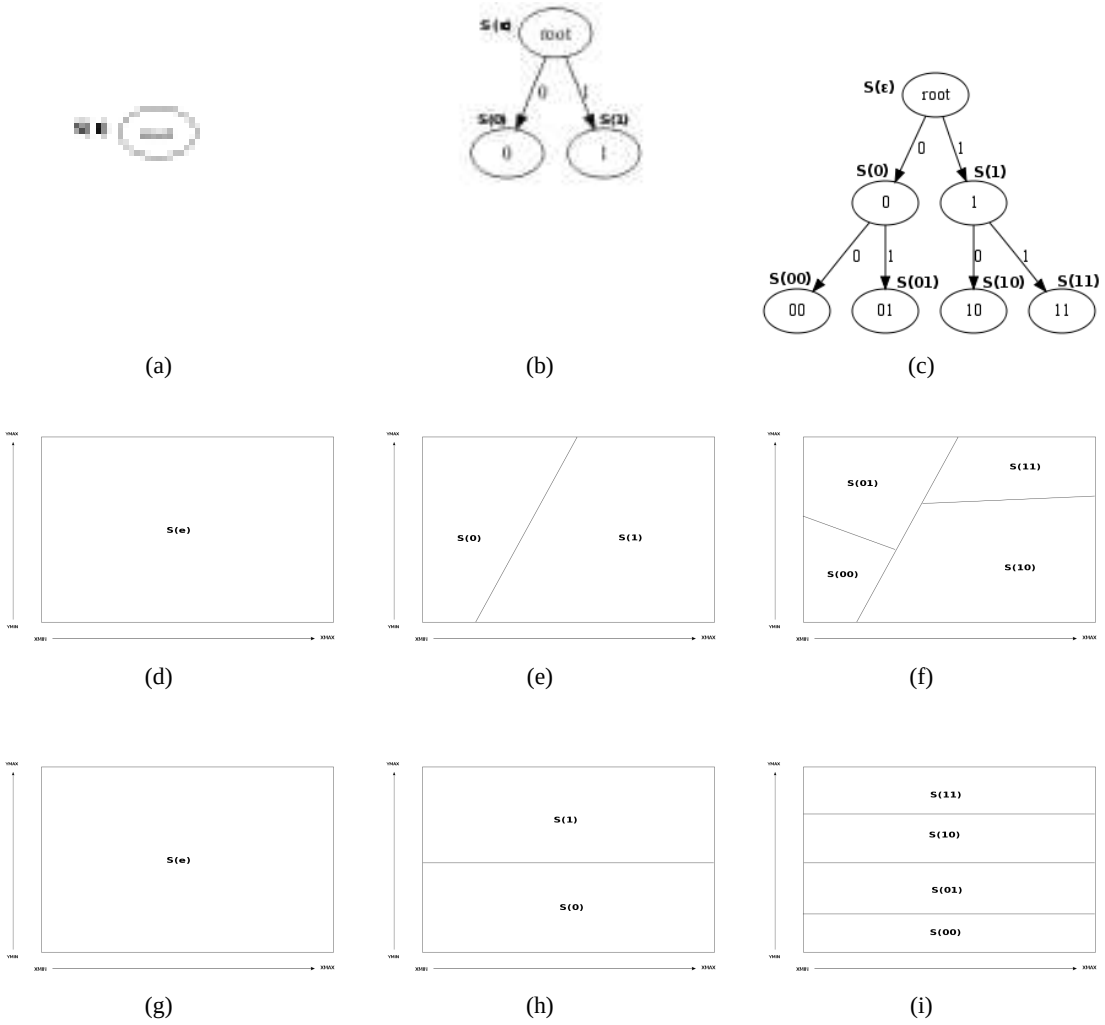


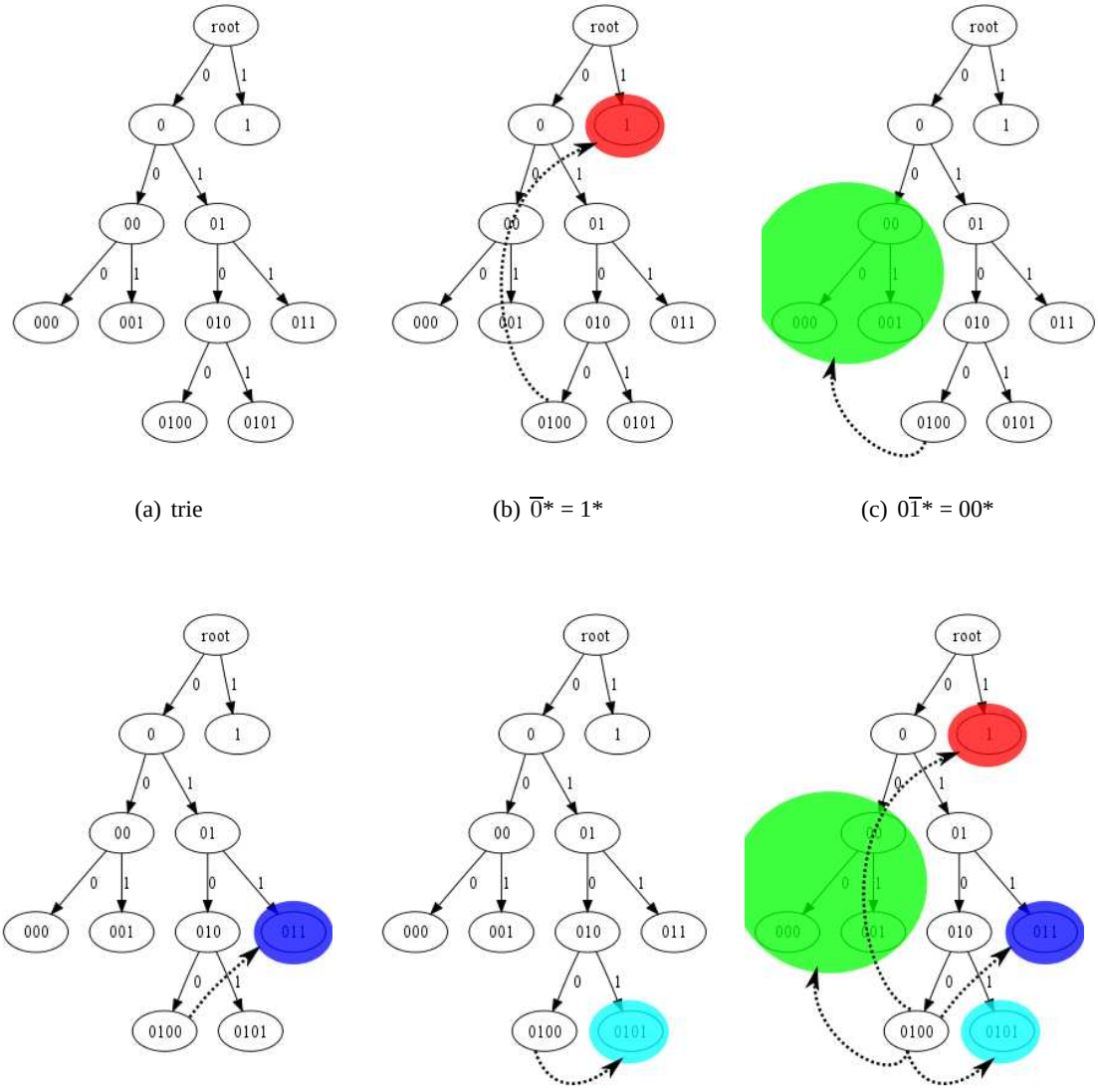
Figure 3.2: The evolution of the trie and two possible space partitionings (a–c). Two possible data partitioning schemes are presented; one on (d–f) and another on (g–i). We denote with S() the data space of each node. The root node is denoted with ϵ.

fact that peers are organized over a trie exactly as they did on PGrid. We also depicted a detailed example on Section 3.1 and Figure 3.1. Moreover, the routing tables are constructed samewise as on PGrid. For clarification purposes we present on Figure 3.3 the construction the routing table of peer 0100.

Having now explained how the peers organize over a trie and how the routing tables are constructed we are ready to describe how to Search Algorithms works. More formally, given a range query r and the peer p who asks it the Search Algorithm returns all the peers that their data space $S(p)$ is intersected r, i.e. $S(p) \neq \emptyset$.

Now let's see the inner details of the Search Algorithm itself. The notation used is the same as the one earlier introduced when describing the Route Algorithm on Section 2.3.3.

Then apply recursively:



(a) trie (b) $\overline{0}$* = 1* (c) 0$\overline{1}$* = 00*

(d) 01$\overline{0}$* = 011* (e) 010$\overline{0}$* = 0101*

Figure 3.3: Constructing the routing table of peer 0100. On (a) we present the overall trie. On (b) we invert the first bit of peer 0100 in order to show that peer 0100 has to have a reference link onto a leaf of the subtrie $\overline{0}$, i.e. in our case onto peer 1. Samewise for (c), (d) and (e). On (f) we present all the neighbors which peer 0100 has links to.

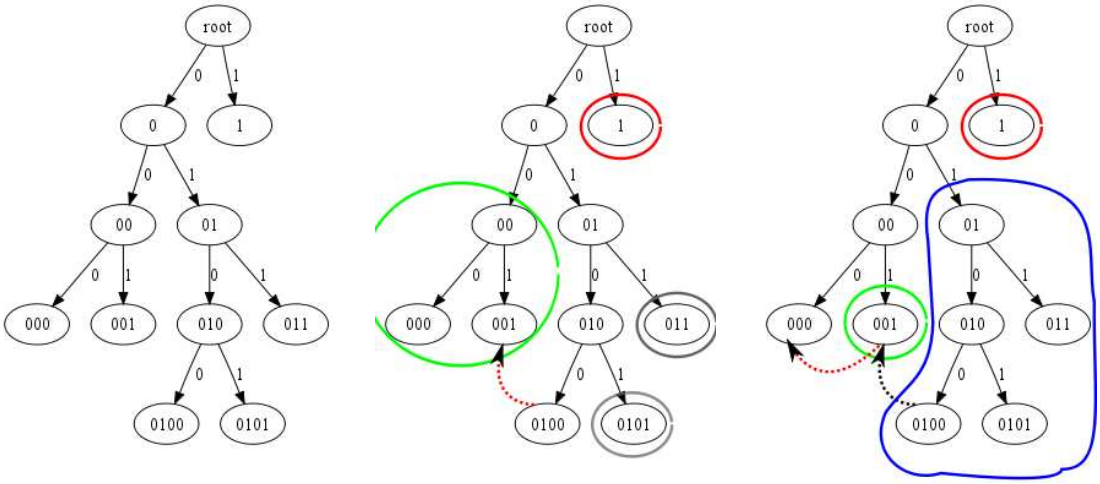
Search(Peer p**, Range** r**, int l) {**

if($S(p) \cap r \neq \emptyset$ **)**
answerLocally(r**);**

foreach(prefix x **of** p**, such that** $|x| \geq l$**)**

if($S(x \cdot \overline{\textbf{bit}(p,|x|)}) \cap r \neq \emptyset$ **)**
Search($L_p[x]$**,** r**,** $|x|+1$**);**

}



(a) (b) (c)

Figure 3.4: Example of Search Algorithm. Peer 010 searches for the range query r. The subtries for which a peer in question should have a pointer to a leaf (at least) are circumscribed with circles. Assume that peer 0100 asks a range query r which for sake of simplicity can be answered only be peer 000. Peer 0100 cannot answer the query and therefore forwards it to its neighbor 001. 001 cannot also answer the query and therefore forwards it to its neighbor 000 which can now fully answer the query. The series of executions of the Search Algorithm is the following: Search(0100,r,1) $\rightarrow$ Search(001,r,2).

The aforementioned algorithm is quite simple. It takes as arguments the peer p which asks the generalized query r and an integer l. l denotes the maximum depth of the trie that the query has visited so far and is used as a boundary limit in order to avoid cycles when forwarding queries. The peer that initiates the query sets l equal to 0 and each peer that accepts a query increases l by one when reforwards the query. If the key space $S(p)$ of the peer which receives the query is intersected with the query r then the peer answers the query. Then the peers checks its routing table to find a neighbor that its key space is also intersected with the query. If this is the case then it forwards the query to it with incremented by one the boundary limit l so that the neighbor won't forward the query back to it. We should mention here that if the query can be forwarded to multiple neighbors then only one of them is chosen (uniformly) for load balancing purposes.

Each peer has at least one pointer to the other side of the trie. Therefore a hop can traverse half of the maximum distance among the peers. Or in other words the Shower algorithm at each call halves the number of the undiscovered peers by two. Therefore the number of the hops needed to answer an abstract query is $O(logn + k)$, where n is the number of the peers and k is the number of the reported items.

Note that a key may be returned multiple times during a search. The is obviously a bug but a minor one.

On Figure 3.4 we present an example of peer 0100 searching a range r over the trie of Figure 3.3.

Later on Chapters 4 and 5 we will present two realistic examples of the algorithm's usage for two novel distributed data structures.

3.5 Data Updates

GRaSP allows datum updates (insertions/deletions) in a manner similar to PGrid2.3. Assume peer q wants to insert (delete) the key K on the network. Then the peer p stores $A_K(S(p))$. In a more detailed approach what is happening is that peer q executes the Search Algorithm with input the key in question, i.e. Search(Peer q, d, 0). The query is forwarded until it reaches one or more peers that are responsible for the data space that includes the candidate key. A similar process is followed for a datum removal. A datum update can be approached as a removal and insertion view a double-execution of the Search Algorithm — once for the removal of the obsolete datum and one for the insertion of the new one.

The above process would be very expensive for a new peer joining the network because it would have to repeat it for every datum in its disposal in order to distribute them into the network. Therefore we introduce the notion of *batch updates*. Now the new peer p passes a collection of keys into the Search Algorithm instead of a Range.

3.6 Overview of customization steps for GRaSP

All-in-all it is very easy to construct a new protocol using GRaSP. The only issues one has to take care to customize a new protocol is setting up the following abstract parameters.

- Specify the form of the generalized range query: r
  eg is it a point query? a 1-dimensional range query? an d-dimensional rectangular range query? (see Section 1 for the definition of the Generalized Search Problem and some instances of it)
- Specify (hierarchical) space partitioning (some specific conditions should guilt): $S(p)$
  (see Section 3.2)
- Specify when a peer can answer a generalized range query: $S(p) \cap r$
  (see Section 3.4)
- Choose a bootstrapping algorithm
  (see Section 3.3)

Obviously, all the aforementioned parameters have to been chosen beforehand running a customized protocol of GRaSP.

3.7 Sum Up

To sum up we have provided a framework that borrows the trie overlay of PGrid and generalized the Shower Algorithm into the Search Algorithm in order to handle any generalized range query in any dimensions. We have also given instructions on the Space Partitioning tactic that should be followed along with a few conditions that should be full-filled.

On the following two chapters we implement two protocols based on GRaSP in order to exhibit and evaluate it: Multidimensional Range Search (MDRS) and Three-sided Range Search (3SIDED).

MDRS can handle multidimensional rectangular range queries and 3SIDED can handle multidimensional 3-sided range queries.

Chapter 4

MDRS

In the previous chapter we considered the GRaSP framework. We saw that GRaSP facilitates the construction of new distributed data structures by hiding the technical difficulties emerged, such as the middleware. In this chapter we customize GRaSP in order to craft a novel protocol that can handle multi-dimensional rectangle range queries over rectangle keys. We call it *Multidimensional Range Search (MDRS)*.

In order to define MDRS more formally we instantiate the Generalized Range Search Problem earlier defined on Chapter 1. Now, the search space is $U = [0, 1]^d$, where d is the number of the dimensions. The key space and the range space contain d-dimensional rectangles, i.e. $\mathcal{K} = \mathcal{R} = \{d\text{-dimensional rectangles}\}$.

A d-dimensional rectangle can be depicted as a hyper-rectangle on d dimensions bounded on left-bottom by the point $P_{min} = (P1_{min}, P2_{min}, \ldots, Pd_{min})$ and on top-right by the point $P_{max} = (P1_{max}, P2_{max}, \ldots, Pd_{max})$.

A typical 2-dimensional rectangle range query is depicted on Figure 4.1.

For an exemplary application consider a Geographical Information Systems (GIS) package. GIS is an information system for capturing, storing, analyzing, managing and presenting data which are spatially referenced (linked to location)[1]. An exemplary function would be to locate all the parks that are located on a user-specified rectangle over a map which only includes all the parks of Greece. Here the map of the earth would be the search space U, the rectangle would be the range

[1]Definition of GIS taken from http://en.wikipedia.org/wiki/GIS.

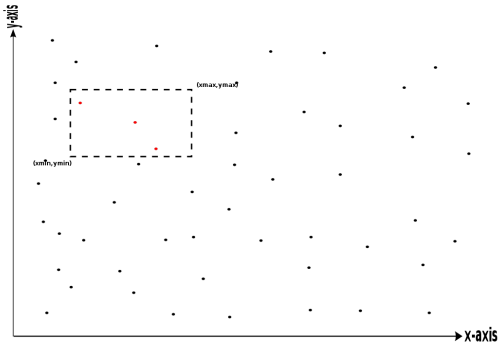


Figure 4.1: 2-D Range Query

space R and Greece would be the data space K. This is a typical example of a *2-dimensional rectangle range query*.

According to the proposed steps mentioned on Section 3.6 for the customization of GRaSP we have structured the organization of the chapter. Initially on Section 4.1 we specify the space partitioning algorithm. Next, on Section 4.2, we describe when a peer can answer an rectangle range query. We conclude with an example of the Search Algorithm applied over a MDRS trie.

4.1 Hierarchical Space Partitioning

In order to achieve load balancing we want to exploit the nature of the problem, i.e. the fact that the queries are rectangle and can happen anywhere on the space. Therefore we would like to split the space into rectangles aligned to x and y dimensions. The obvious solution is following the idea of k-d trees, i.e. $S(peerID)$ is splitted along dimension $|peerID| \mod d$, where $|id|$ is the length of the bitstring id. Or other words we split the key space of a peer in a Round-Robin manner. For example in the case of $d = 2$ the x and y splitting dimensions are interchanged at each level of the trie.

Obviously there aren't any peers holding the same keys ($S(x \cdot 0) \cap S(x \cdot 1) = \emptyset$). Therefore each key is inserted only once into the network. Later on on Section 6.2.1 we will more say that replication equals 1.

4.1.1 Example

In order to depict the aforementioned sceptic we present respectively on Figure 4.1.1 and Figure 4.3 an in-depth example of a series of node joins and the respective space partitionings for a 2-dimensional key space. The evolution of the trie is progressing according to what we have said on Section 3.1. For the rest of the chapter (and even the thesis) we will stick with MDRS to two dimensions for the sake of simplicity. Any generalizations to more dimensions are self-intuitive.

For the figures that contain tries (on this and next chapter) the notation used on each node follows the template:

$$peerID : [P_{min} - P_{max}]DIM = splitcoord$$

where:

$peerID$ is the peerID of the node (we use the notation $root$ for the root node instead of the empty string ϵ).

[$P_{min}P_{max}$] are the two boundary points which define the rectangle key space for which the peer is responsible for. This is actually the key space $S()$ of the peer with peerID$= PeerID$.

$DIM = splitcoord$ DIM shows the dimension along which the key space of this node (with PeerID$= PeerID$) will be splitted when a new peer becomes its child (on bootstrapping). Typical values for DIM are the following. If the split dimension is along the y-dimension then DIM equals $YDIM$. Else, if the split direction is along the x-dimension the DIM equals $XDIM$. $splitcoord$ is the coordinate on the DIM-dimension that the key space of this node will be splitted when a new peer becomes its child (on bootstrapping). Obviously, the value of $DIM = splitcoord$ for the root node is dummy without any sense.

For the case of MDRS we to make the following remarks. $YDIM$ and $XDIM$ are interchanged at each level of the trie. Also, the coordinates $XDIM = YDIM = \{$the half of the parental DIM$\}$. Moreover. two sibling nodes have the same $DIM = splitcoord$ value.

Next to each leaf we list its keys and routing table. Obviously the inner (virtual) nodes have neither keys nor routing tables. Each edge of the trie is annotated with 0 or 1.

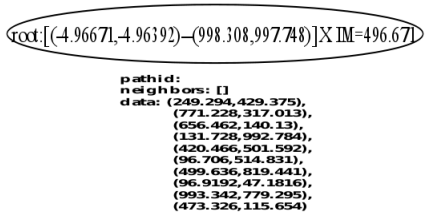
4.2 Orthogonal Range Searching

Having defined the query type r as a rectangle and the key space $S(p)$ of a peer p as a rectangle area of a set of rectangles (and postponed bootstrapping algorithm for Section 6.3.3) the only thing left to fully specify MDRS is checking when a peer can answer a query. *A peer can answer a query r if its key space $S(p)$ is intersected with it, i.e. if $S(p) \cap r$.* If this is the case then we want to find all the keys in $S(p)$ that intersect to r and return them as answers to the query. On other words we want to check if a rectangle intersects a set of rectangles. The core of this problem is checking if two rectangles, let's call them A and B, intersect. This is the case if (a) the boundary top-right point of A dominates (i.e. is more-or-equal) the boundary bottom-left of B and (b) the boundary top-right point of B dominates the boundary bottom-left point of the A.

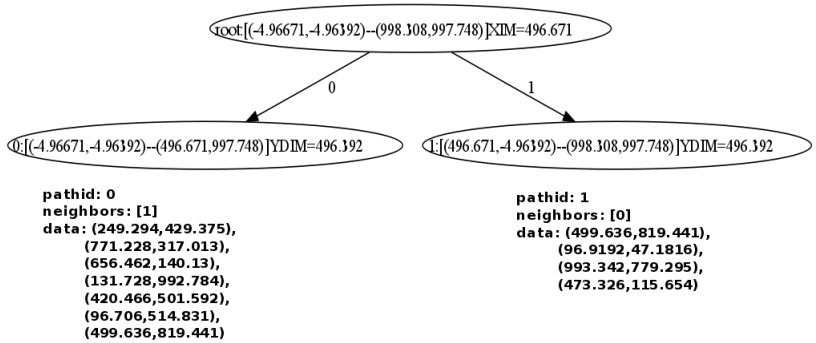
4.2.1 Example

Here we present an exemplary application of the Search Algorithm (see Section 3.4) over the MDRS trie of Figure 4.1.1(e).

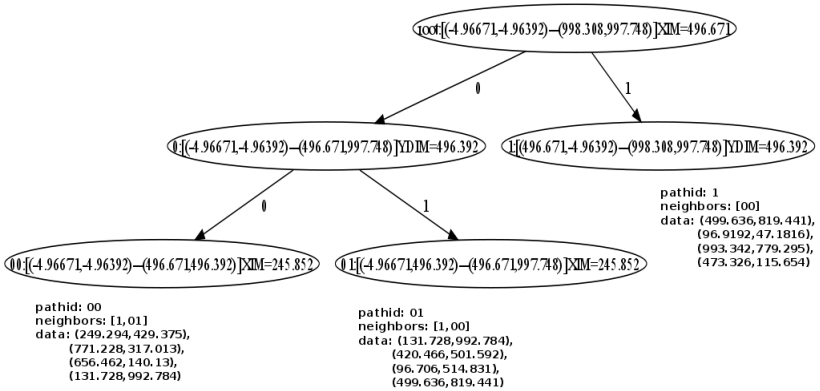
Look at Figure 4.2.1. Assume node 011 asks the range query r with $P_{min} \equiv (x_{min}, y_{min}) = (0.000000, 0.000000)$ and $P_{max} \equiv (x_{max}, y_{max}) = (800.631002, 176.698761)$. Actually this is a real 3-sided query ($(x_{min}, x_{max}, y_{max}) = (0.000000, 800.631002, 176.698761)$) used in our experiments on Section 6.3. Node 011 initially checks if itself can answer part of the query. This is the case and therefore it answers part of the query locally (answerLocally(r) of Search Algorithm). Afterwards if traverses its routing table (the loop of Search Algorithm) and checks if any of its neighbors can answer part of the query too or if at least any neighbor is closest to the query than itself. This is the case with neighbors 1 and 00. Therefore node 011 forwards them the query r (the inner recursive call of Search of Search Algorithm). Recursively the same process is carried out on nodes 1 and 00. Note that node 00 now doesn't forward the query to node 011 again because of the parameter l of the Search Algorithm.



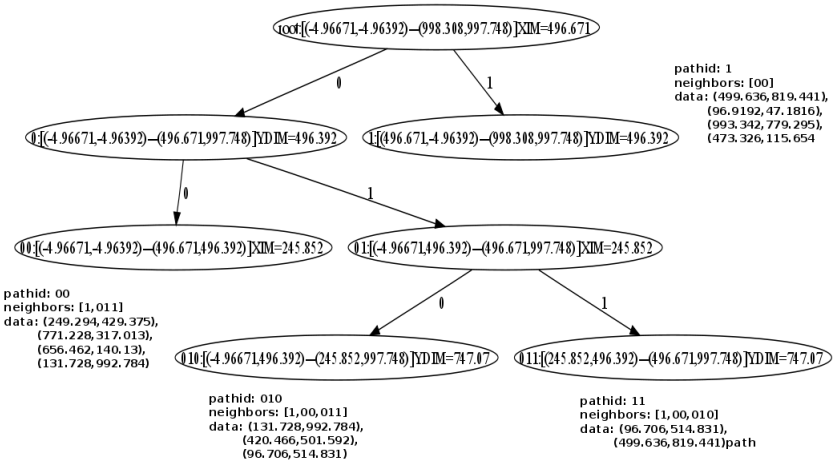
(a) number of nodes=1: {”}



(b) number of nodes=2 {’0’,’1’}

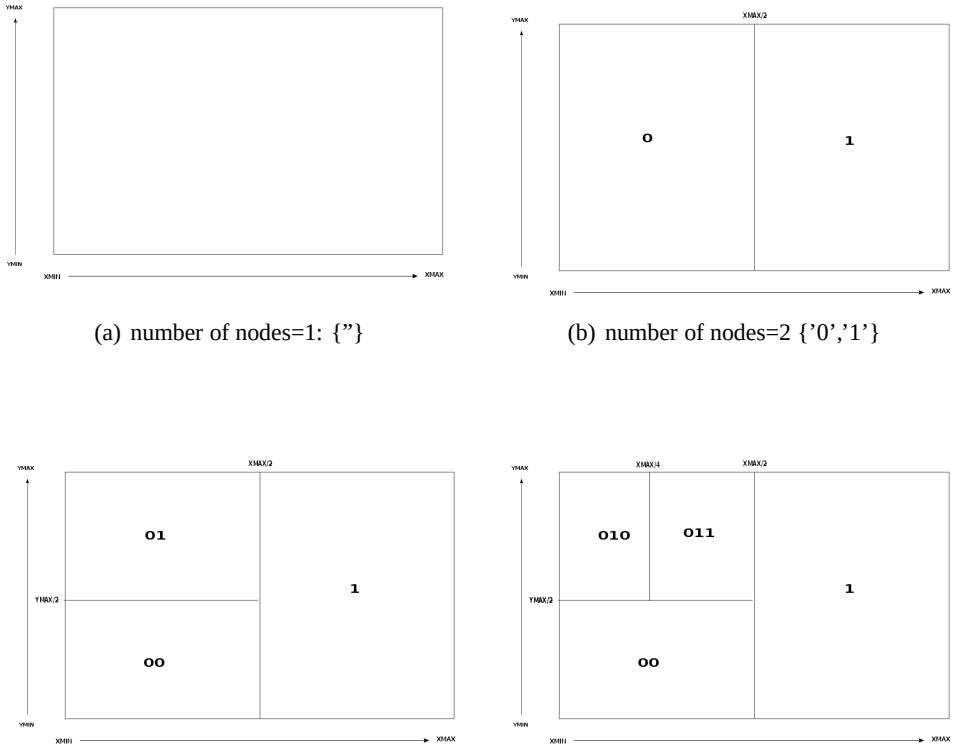


(c) number of nodes=3 {’00’,’01’,’1’}



(d) number of nodes=3 {’00’,’010’,’011’,’1’}

Figure 4.2: Exemplary evolution of the MDRS trie on 2-D while new nodes join arrive.



(a) number of nodes=1: {”}

(b) number of nodes=2 {’0’,’1’}

(c) number of nodes=3 {’0’,’10’,’11’}

(d) 3number of nodes=3 {’00’,’01’,’1’}

Figure 4.3: Exemplary evolution of the 2-dimensional MDRS topology while new nodes join arrive. There is a 1-to-1 mapping between this figure and Figure 4.1.1.

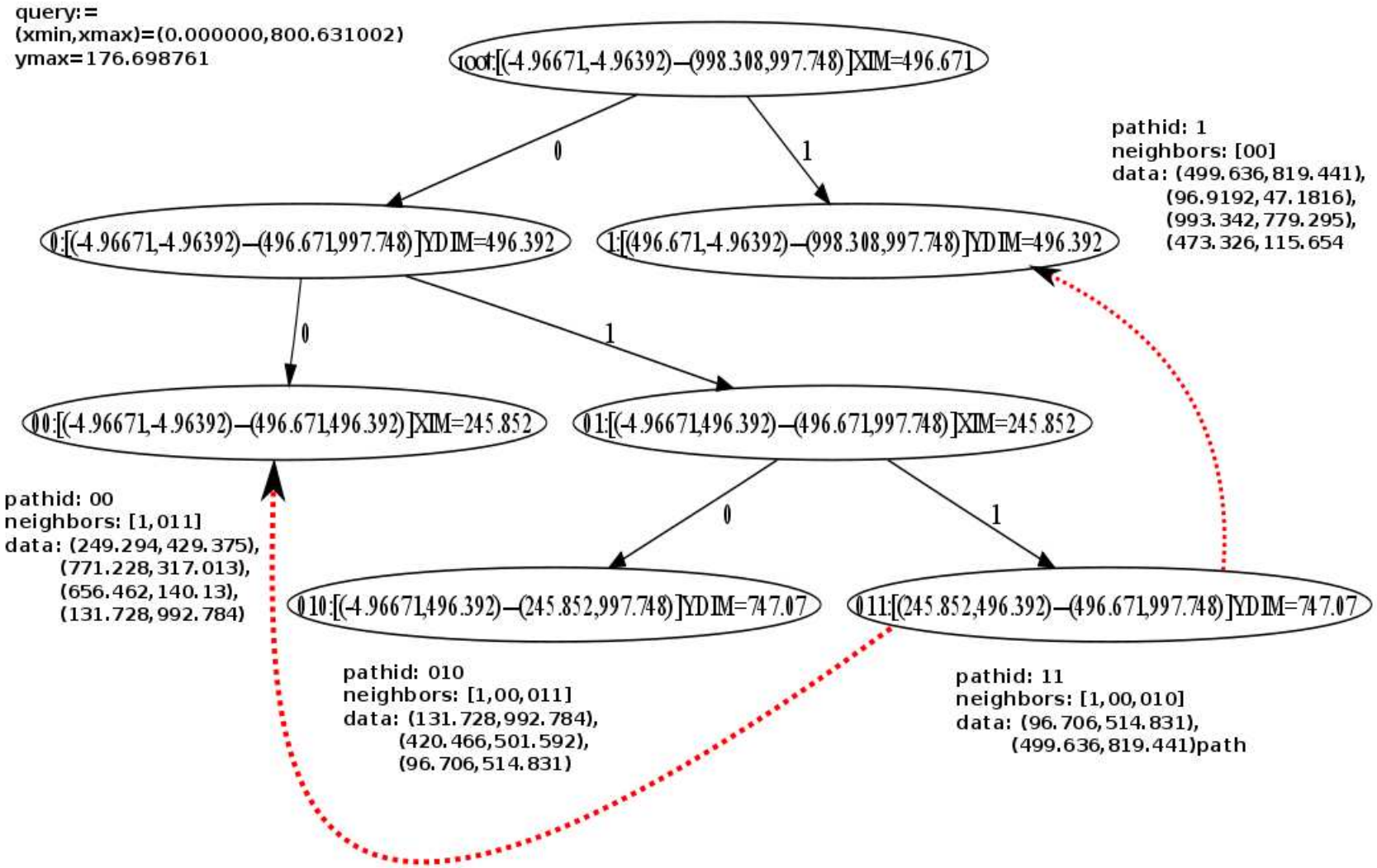


Figure 4.4: Example of answering a 2-dimensional orthogonal range query over the MDRS trie. The trie is the one earlier presented on Figure 4.1.1(e).

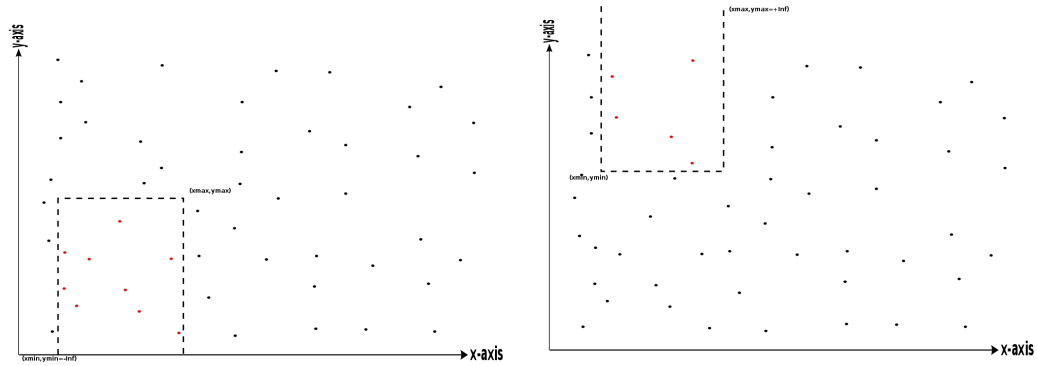
Chapter 5

3SIDED

In Chapter 3 we presented GRaSP, a framework for constructing distributed data structures that can tackle the generalized range search problem over a P2P network. In the previous Chapter 4 we customized GRaSP and constructed MDRS, a protocol that can handle multi-dimensional rectangular range queries. Likewise, in this chapter we will customized GRaSP and instantiate 3SIDED, a protocol that can answer multi-dimensional 3-sided range queries. The process followed to accomplish this is similar to the one followed on the construction of MDRS.

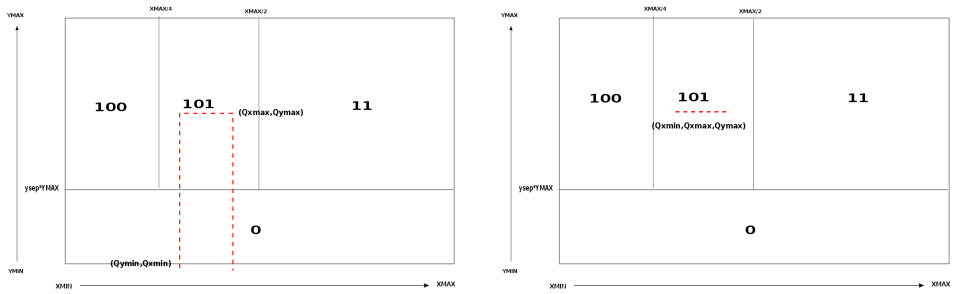
Let's define more formally the 3-sided Range Search Problem. For sake of simplicity we will stick to 2 dimensions — any generalizations are self-intuitive. The keys are points. A *3-sided range query* is a degenerated rectangle range query. More precisely, going back to the definition of the rectangle on the beginning of Chapter 4 we set $d = 2$ and $P2_{min} = 0$. On this case the 3-sided query is *bounded on top*. Alternatively, if $P2_{max} = \infty$ then the 3-sided range query is *bounded on bottom*. Let's relax the notation in order to make things simpler and represent the two dimensions with the familiar x-axis and y-axis. Then a 3-sided range query bounded on bottom is equivalently defined by the triplet of coordinates $x_{min}, x_{max}, y_{min}$ and a 3-sided range query bounded on top by the triplet of coordinates $x_{min}, x_{max}, y_{max}$. A typical 3-sided range query bounded on top is presented on Figure 5.1(a) and another one bounded on bottom is presented on Figure 5.1(b). For the rest of the chapter (and the thesis) we will stick to 3-sided range queries bound on top.

3SIDED uses and alternative representation for the keys and queries. Here we will use the notation earlier introduced when defining the Generalized Range Search Problem earlier defined on



(a) 3-sided Query (bounded on top) (b) 3-sided Query (bounded on bottom)

Figure 5.1: Example of 3-sided queries. On (a) we have 3-sided query bound on top and on (b) a 3-sided query bounded on bottom.



(a) 3sided Query (b) 3sided Query (reduced form)

Figure 5.2: 3-sided range query representation and its equivalent reduced form as used on 3SIDED.

Chapter 1. First of all we define the search space as $U = [0, 1]^d$, where d is the number of the dimensions. Each key is depicted as an upward ray, i.e. $\mathcal{K} = \{\text{upward rays}\}$. Each 3-sided range query is depicted as an horizontal segment, i.e. $\mathcal{R} = \{\text{horizontal segments}\}$. On Figure 5.6 we present a typical 2-dimensional 3-sided range query and its equivalent horizontal segment reduction used by 3SIDED.

Typical applications of the 3-sided range search problem are found when working with financial data. Assume for example that on a 2-dimensional space we the x-axis t represents the time progress (in days of month) and the y-axis s represents the sales of products (in dollars). We want to find all the products that had high sales during Christmas. Then the predicate would be $s > 100\$$ and $25 <= t <= 31$ on December. This an example of a *3-sided query bounded on bottom*. Alternatively we could find low sales and this would be a *3-sided query bounded on top*.

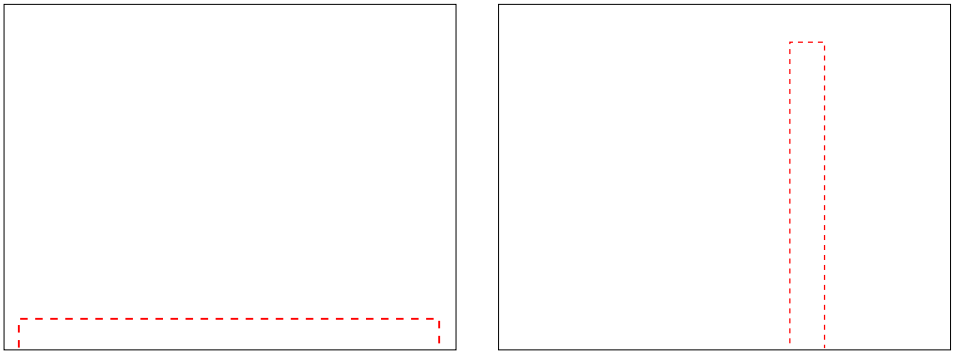
According to the guidelines earlier provided on Section 3.6 in order to fully specify the 3SIDED protocol we have to specify the format (shape and dimensionality) of the data and query, when a peer can answer a query and the hierarchical space portioning scheme. Again, we postpone the choice of the bootstrapping algorithm for Section 6.3.3. Accordingly we have organized the structure of the chapter. Namely, on Section 5.1 we specify the space partitioning followed by 3SIDED and the reason motivated it. Afterwards on Section 5.2 we specify when a peer can answer a query.

A strong emphasis should be given to the fact that, in our knowledge, there isn't any previous work on P2P that tackles the multi-dimensional 3-sided range search problem, i.e. there isn't any network that can handle 3-sided range queries. Our work is pioneer and depicts the usefulness of GRaSP once again.

5.1 Hierarchical Space Partitioning

Handling 3-sided queries is a very demanding problem. The most prominent reason is that all the queries hit peers low on the key space. This is especially obvious for the low-and-wide 3-sided queries. Moreover, if a 3-sided query is tall-and-narrow many peers will be hitted. On both cases each peer will return a small part of the answer. Typical examples of both queries are depicted on Figure 5.3.

On Section 3.2 GRaSP proposes that in order to ameliorate the load balancing problem the Space Partitioning scheme should be adapted to the nature of the problem, i.e. to the difficult queries



(a) low-and-wide (b) tall-and-narrow

Figure 5.3: Difficult 3-sided queries. It's obvious their particularities and the problems they pose.

depicted on Figure 5.3. Therefore we want many peers on the bottom of the space and less on the top. The lower peers should hold low-and-wide areas and and the upper ones tall-and-narrow. This can be achieved by adding redundancy low on the key space and splitting the key space in rectangles, low-and-wide on bottom and tall-and-narrow on top as we mentioned before. This idea can be effectuated by adopting the following algorithm. The space partitioning algorithm adopted by 3SIDED contains two type of splits, one horizontal and one vertical. When a key space is splitted vertically then it is splitted into the middle of the x-axis (remember that for sake of simplicity we just refer two 2 dimensions). Therefore two equal volumed key spaces result. On the case of the horizontal split the key space is splitted on the y-axis not on the middle but on a the Y_{SEP} percentage of its height. Y_{SEP} is a user specified parameter and controls the overlapping between the key spaces of the peers. The higher Y_{SEP} is the higher the overall overlapping is. Obviously $Y_{SEP} \in \{0, 1\}$. Assume a new peer p wants to join the network and becomes child of the parent (bootstrap) node b. If key space of b was resulted from an horizontal split and the last bit of b is 0 then its key space is also splitted horizontally. If it ended on 1 then it is splitted vertically. On the opposite case where the key space of b resulted from vertical split then it is also resplitted vertically. Section 3.1 tells us which part of the key space of b each newly created child gets. This ideas are presented on a more formal manner on the algorithm known as *3sidedHashing* on Algorithm 1.

Assume we want calculate the key space of peer with PeerID bs, i.e. the value of $S(bs)$. Also assume that the search space U is bounded respectively by the bottom-left and top-right points $P_{min} = \{X_{MIN}, Y_{MIN}\}$ and $P_{max} = \{X_{MAX}, Y_{MAX}\}$. Also assume that we have set Y_{SEP}. Then given the arguments as input to the hashing algorithm then 3sidedHashing returns the followings:

$p_{min} = \{x_{min}, x_{max}\}$**,** $p_{max} = \{y_{min}, y_{max}\}$ The key space of peer bs is bounded respectively by the bottom-left and top-right points p_{min} and p_{max}.

y_{sep} is the coordinate on the y-dimension that the key space of peer bs will be further splitted if it adopts a peer. This may be equal to y_{min} in case its key space will be vertically splitted.

Just to refresh the obvious we mention here where the 3sidedHashing Algorithm is used. This is the case when a new peer joins the network (see Section 3.1) the peer that will adopt it (bootstrap/parent peer) should split its key space into two (overlapping or not) subregions and each one be given to each peer. The 3sidedHashing Algorithm tells us where to split the key space of the parent peer.

Algorithm 1 3sidedHashing (X_{MIN},Y_{MIN},X_{MAX} and Y_{MAX} are the boundaries of the overall data space).

1: **function** 3SIDEDHASHING(X_{MIN},Y_{MIN},X_{MAX},Y_{MAX},Y_{SEP},bs)
2: $x_{min} \leftarrow X_{MIN}$
3: $x_{max} \leftarrow X_{MAX}$
4: $y_{min} \leftarrow Y_{MIN}$
5: $y_{sep} \leftarrow Y_{MIN}$
6: $y_{max} \leftarrow Y_{MAX}$
7: $hsplit \leftarrow True$
8: **for all** $i \leftarrow 1, size(bs)$ **do**
9: $b \leftarrow bs[bs_i]$
10: **if** $hsplit = True$ **then**
11: $y_{middle} \leftarrow y_{min} + Y_{SEP} * (y_{max} - y_{min})$
12: **if** $bit = 0$ **then** $y_{max} \leftarrow y_{middle}$
13: **else**
14: $y_{sep} \leftarrow y_{middle}$
15: $hsplit \leftarrow False$
16: **end if**
17: **else**
18: $x_{middle} \leftarrow x_{min} + 0.5 * (x_{max} - x_{min})$
19: **if** $bit = 0$ **then**
20: $x_{max} \leftarrow x_{middle}$
21: **else**
22: $x_{min} \leftarrow x_{middle}$
23: **end if**
24: **end if**
25: **end for**
26: return [x_{min},x_{max},y_{min},y_{sep},y_{max}]
27: **end function**

Note that on 3SIDED we don't pose the restriction $S(x \cdot 0) \cap S(x \cdot 1) = \emptyset$ as we did on MDRS on Section 4.1). Therefore we can introduce replication, i.e. overlapping key spaces, into the network. This is useful as we have already explained for load balancing reasons.

5.1.1 Example

A detailed example (for $Y_{SEP} = 0.3$) of a series of peer joins and the respective evolution of the trie and the topology is depicted on Figure 5.1.1 and Figure 5.5. Actually, this is a realistic case borrowed from our experiments on Section 6.3. The notation used is consistent to the notation earlier introduced and used on Section 4.1.1.

5.2 3-sided Range Searching

Having defined the query r as an horizontal segment and the key space $S(p)$ of peer p as a rectangular area that contains a set of upward rays (and postponed bootstrapping algorithm for Section 6.3.3) the only thing left to fully specify 3SIDED is checking when a peer can answer a query. *A peer can answer a query if its sub-key-space bounded on bottom-left by the point $p_{min} = \{x_{min}, y_{sep}\}$ and on top-right by the point $p_{max} = \{x_{max}, y_{max}\}$ is intersected the query horizontal segment. The aforementioned coordinates results from the execution of the 3sidedHashing Algorithm with input $PeerID = p$.* Note the distinction between the key space of a peer, i.e. the area for which a peers holds all the inlaid keys, versus the space of a peer for which the peer is responsible for answering queries.

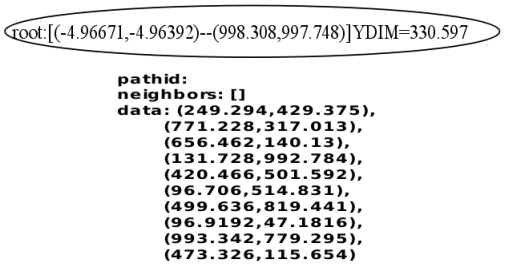
Now that we know if a peer can answer a query we want to get all its keys that answer the query. A brute force method with linear time complexity would be to compare all the upward ray keys of the peer with the horizontal segment of the query.

On Figure 5.6(b) we present an horizontal segment query intersected with 3 upward ray keys (we also depict the original 3-sided range query and keys for clarification purposes). On Figure 5.6(a) we see that responsible for answering the 3-sided range query (bounded on top) is peer 010. Peer 010 holds keys a, b and c. Only keys a and b answer the query. The answer keys are colored red. On Figure 5.6(b) we see an equivalent picture where the query is an horizontal segment and the keys are upward rays. Again the upwards rays a and be answer the query.

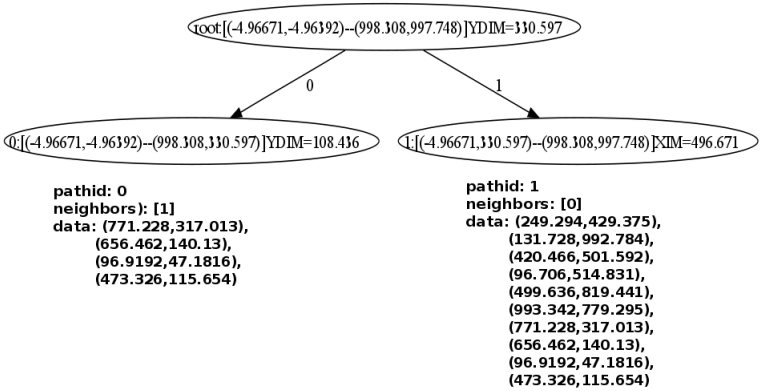
An important notice is the following. Practically the query can be fully answered by peers 010 and 101 since their key spaces are intersected the query. But from what we have said at the beginning of this section it's obvious that only peer 010 is intersected the query. Therefore is peer 101 receives the query in question it will discard it or forward it to a neighbor.

5.2.1 Example

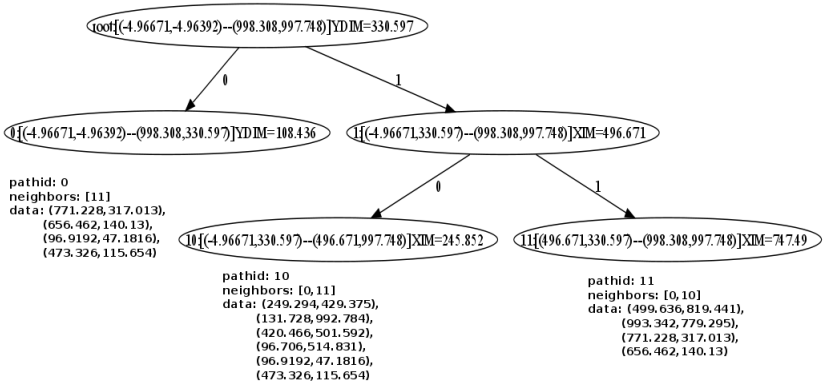
Here we tersely present an exemplary application of the Search Algorithm (see Section 3.4) on Figure 5.2.1. The steps followed are similar to the respective case of the MDRS protocol on Section 4.2.1 and therefore we avoid plagiarism.



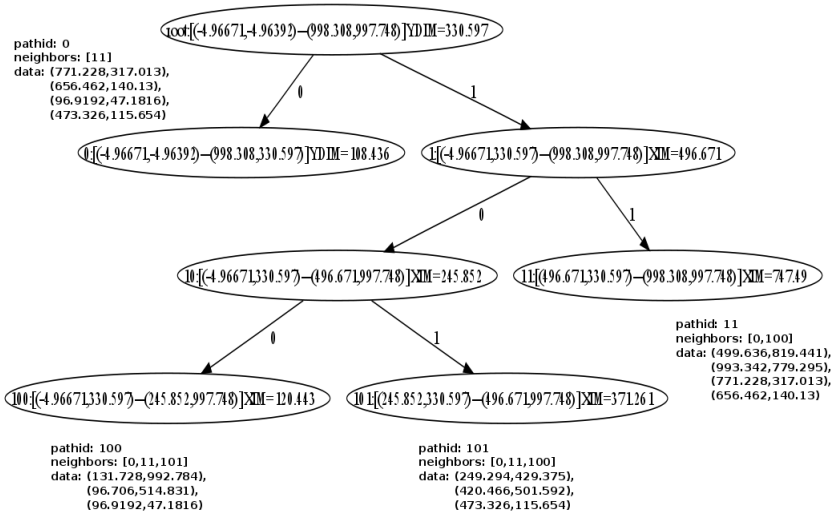
(a) number of nodes=1: {”}



(b) number of nodes=2 {’0’,’1’}

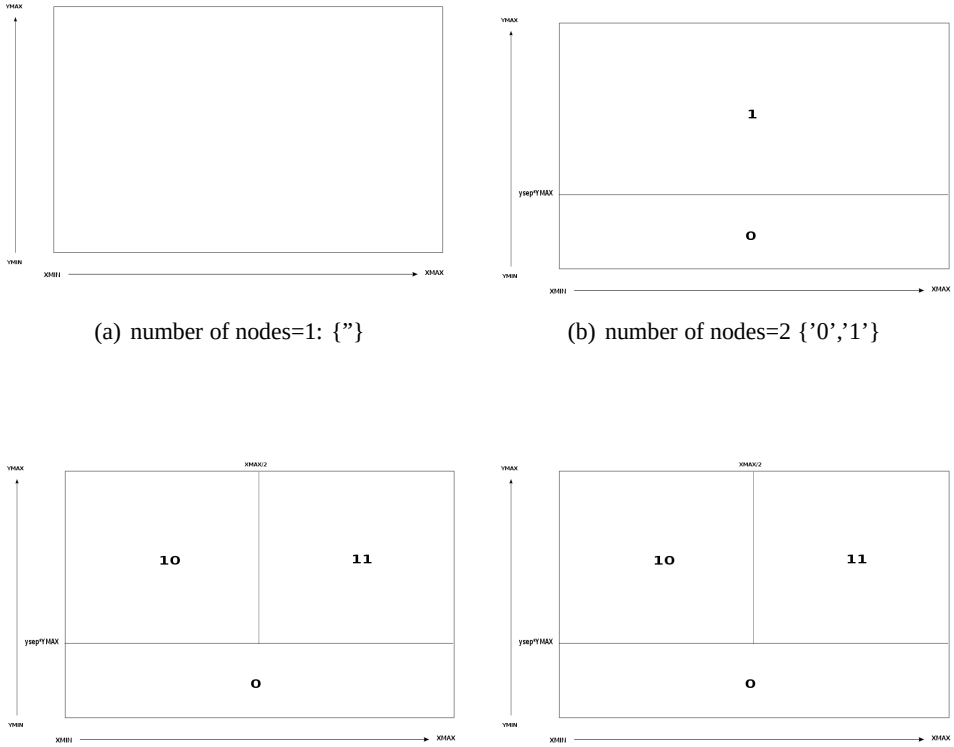


(d) number of nodes=3 {’0’,’10’,’11’}



(e) 3number of nodes=3 {’00’,’01’,’1’}

Figure 5.4: Exemplary evolution of the 3SIDED trie on 2-D while new nodes join arrive.



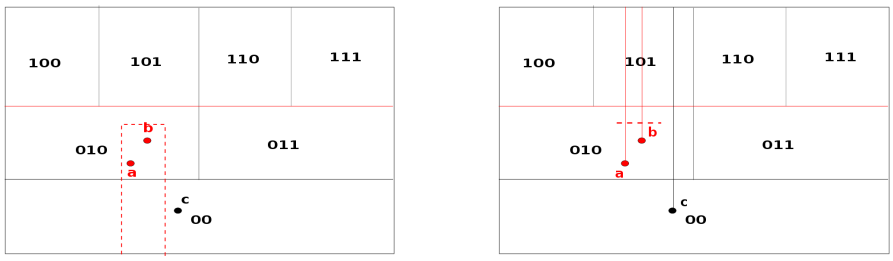
(a) number of nodes=1: {”}

(b) number of nodes=2 {’0’,’1’}

(c) number of nodes=3 {’0’,’10’,’11’}

(d) 3number of nodes=3 {’00’,’01’,’1’}

Figure 5.5: Exemplary evolution of the 2-dimensional 3SIDED topology on while new nodes join arrive. There is a 1-to-1 mapping between this figure and Figure 4.1.1.



(a) 3sided Query

(b) 3sided Query (reduced form)

Figure 5.6: Depicting when a peer can asnwer a 3-sided range query on 3SIDED. Practically both peers 010 and 101 can answer the query but on 3SIDED only 010 does so! On (b) keys are upward rays and the 3-sided range query is an horizontal segment. Keys a and b anwer the query and are colored red. Key c doesn’t answer the query and is colored black.

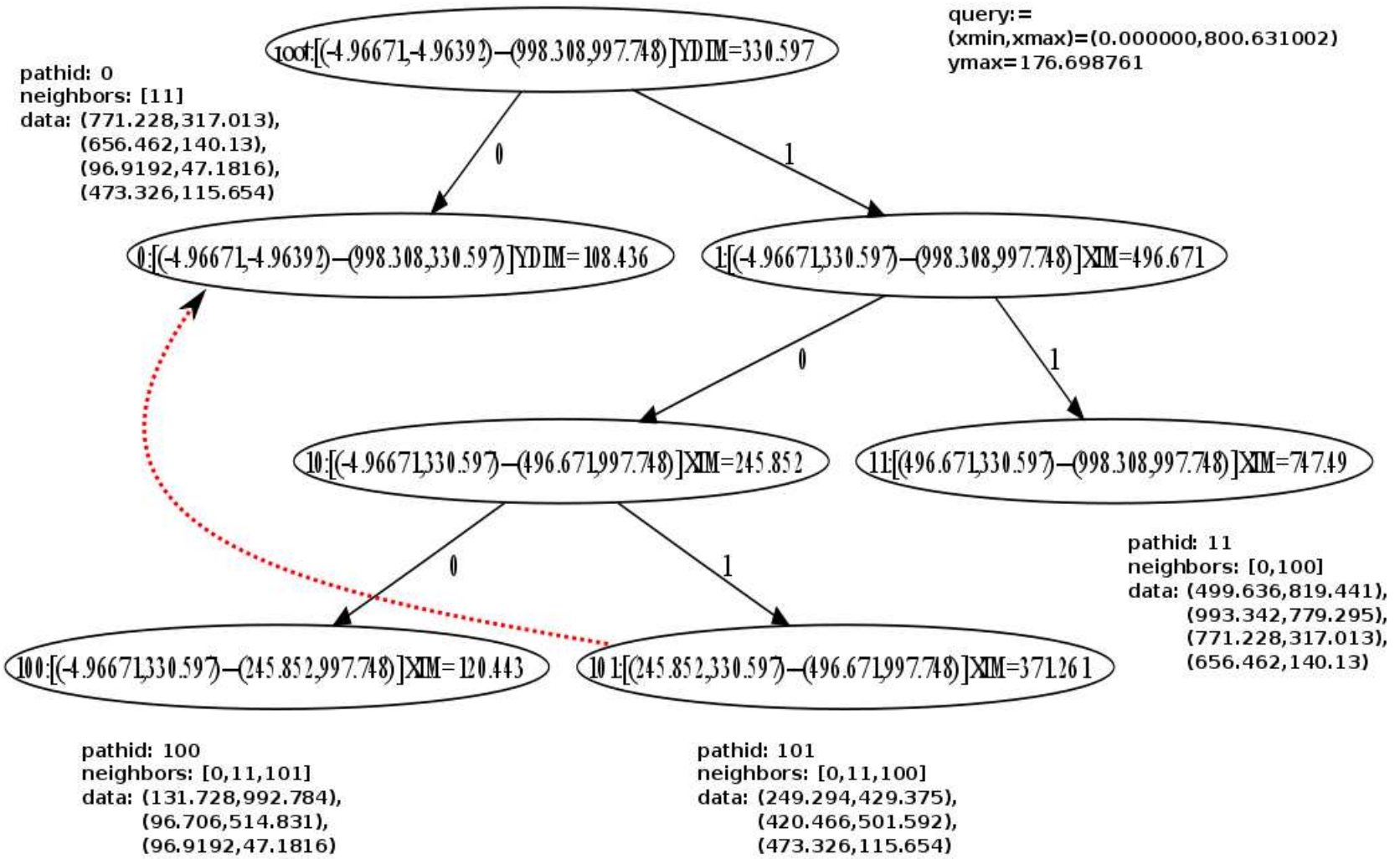


Figure 5.7: Example of answering a 2-dimensional 3-sided range query over the 3SIDED trie. The trie is the one earlier presented on Figure 5.1.1(e).

Chapter 6

Performance Evaluation

In order to evaluate GRaSP we have constructed a fast and scalable simulator called RangeSim-Cpp. This is the mean of the evaluation and is presented on Section 6.1. On Section 6.2 we present the Cost Model on which we have based the evaluation, i.e. the metrics used to evaluate the quality of a network. On Section 6.3 we experiment with 2-dimensional orthogonal range queries over MDRS and 3-sided queries over 3SIDED and MDRS (we regard a 3-sided query as a range query unbounded on a side).

6.1 Simulators

In order to develop protocols over the GRaSP framework we needed to develop a consistent API. This API gives the necessary mechanisms to represent the trie topology of GRaSP, the routing tables, the searching algorithm, the bootstrapping and to customize any Space Partitioning algorithm. Initially we experimented with a Java simulator for P2P called *Peersim*. Due to its lack of efficiency (for reasons that we will justify later on) we have developed a novel C++ simulator, called *RangeSimCpp*.

6.1.1 Peersim

Peersim[33] is a configurable, extendable and self-contained simulator for P2P protocols in Java. It can support the simulation of large networks and the processing of many queries. It can also support dynamic protocols, where node additions and removals are happening. Peersims has two modes of operation, the Cycle Based and the Event Driven.

Cycle Based Simulation

The Cycle Based Simulation is a simplest simulation mode. Herein nodes are given the control periodically, in some sequential order. The processing of queries follows a Breadth First Search manner in the following sense; all the queries that are no answered till the current cycle i are either answered locally or forwarded. The ones answered are discarded from the network whereas the ones forwarded are reprocessed in similarly manner on cycle $i + 1$. This process is continued until all the queries are fully answered.

Event Driven Simulation

In the event based model there is not the semantic of the cycle. On the contrary the events (eg the queries in our case) have to be scheduled explicitly and these are the ones which drive the flow of the simulation. Therefore this mode is more realistic but also less efficient. The Event Driven Simulation also supports transport layer simulation versus the Cycle Based Simulation.

In our experiments we have employed the Cycle Based Simulation mode of Peersim on static networks. Unfortunately the memory and processing demands of Peersim have not been satisfactory and therefore a new simulator has been developed from the scratch in C++ which is working in a similar Cycle Based mode.

6.1.2 RangeSimCpp

The C++ implementation has been called *RangeSimCpp* and is working in a Cycle Based mode but in a Depth First Search manner on the contrary to Peersim. This means that each the query is feeded to the simulator, processed on a cycle-by-cycle manner until it is fully answered. Afterwards it is discarded from the simulator and a new query begins. This process is followed until all queries are fully answered. Obviously only one query is loaded at any time slice. The benefits are obvious and are further discussed on Section 6.1.3.

6.1.3 Comparison

Peersim and RangeSimCpp exhibit some common attributes and some crucial differences. Here we compare the two aforementioned simulators side-by-side in relation to their common Cycle Based operation. First of all Peersim is written in Java whereas RangeSimCpp in C++. Each language provides each pros and cons. Java supports memory garbage collection but is not as fast as C++. On the contrary C++ is not safe from memory leaks but is faster than Java. The most important difference however is the processing algorithm of the queries that each simulator follows; in Peersim we load all the queries in memory and all of them are processed (forwarded or answered) on each cycle (Breadth First processing) whereas in RangeSimCpp we load each query in memory, we process each until is fully answered and afterwards we load the next one and so on (Depth First processing). The subtle difference between the two aforementioned algorithms is that former requires all the queries residing in the memory during the whole duration of the simulation whereas the latter one requires only one query being in the memory at any time. Therefore, according to our experience, Peersim cannot handle efficiently simulations of tens of thousand of queries. For instance if the number of peers is of order of 100K then the memory needs are of order GB and the current CPUs cannot handle it at all. On the contrary RangeSimCpṕs memory needs are of order MB and the time needs are of order of minutes.

6.2 Modeling P2P Network Performance

In order to evaluate GRaSP more general any P2P protocol we borrow some evaluation metrics. Here we mention the most important ones which are the ones used later on our experiments.

6.2.1 Replication

By replication (of data) we define the percentage of the original dataset that is stored in our network. Obviously the replication is greater or equal to one (equal to one when there isn't any). Replication may exist when two or more peers possess same keys; this may be the case for example for load balancing when we want the queries for these keys to be splitted between the two peers.

6.2.2 Fairness Index

Another metric pertinent to the data is the *Fairness Index*[32] which is defined as $FI = \frac{(\sum_{i=1}^{n} x_i)^2}{n \sum_{i=1}^{n} {x_i}^2}$ where x_i is the number of data points of peer i and n is the number of peers. In essence Fairness of Index shows the fairness of the distribution of the data on the peers; i.e. how air are the data distributed among the peers. FI is continues, scale independent, i.e. applies to any network size (even for a few peers only) and is bounded between 0 and 1 — 0 for unfairness (when one peer holds all the data), 1 for equally fairness (when all the peers have equal number of data).

6.2.3 Average per-process Message Traffic, Maximum Throughput

Average per-process Message Traffic and Maximum Throughput are novel metrics initially introduced by Blanas and Samoladas[42]. They studied a few recent P2P networks through d-dimensional orthogonal range queries (PGrid[10], VBI[9], CAN[4], MURK[8]) and concluded through extent simulation that all of them except PGrid don't scale with the number of peers. They emphasized that it not enough measuring the average load per peer in order to reach a verdict for the scalability of a network. What should be measured is the load of the most loaded peer, i.e. the Maximum Throughput.

Let's define now the Average per-process Message Traffic and Maximum Throughput. Assume peer j accepts incoming messages at an arrival rate λ_j, where each message has service demand s. On the other hand this peer has maximum service rate γ_j for an incoming message. Therefore the time required to serve an incoming message is s/γ_j. More generally assume S is a random variable that denotes the distribution of the service demand of the incoming messages. Accordingly define $S_j = S/\gamma_j$ and denote $E[S_j]$ being its expected value.

If P is a set of processes with individual popularities ϕ_p and $m_j(p)$ is the number of messages in process $p \in P$ received by peer j then the *message distribution* of the network is defined as $\mu_j = \frac{\sum_{p \in P} \phi_p m_j(p)}{\gamma_j}$ and the *Average per-process Message Traffic* as $M = \sum_{j=1}^{n} \gamma_j \mu_j$.

The *Maximum Throughput* is defined as $\Lambda_{max} = \frac{1}{E[S] \dot{max}_j \mu_j}$. Obviously, Maximum Throughput is the inverse of the average per-query processing time of the most-loaded peer in the network. In our experiments we have set for simplicity $E[S] = 1$ (i.e. each peer can service the 100% of the incoming messages) and therefore $\Lambda_{max} = 1/max_j \mu_j$. Simply stated it we could say the Maximum Throughput is a metric which measures the load of the most loaded peer or alternatively that is the maximum query rate that the network can sustain indefinitely. Above Λ_{max} some peers may become overload and crash or on the best cast discard messages.

6.3 Experiments

In order to evaluate GRaSP we have implemented the MDRS and 3SIDED protocols (see Chapters 4 and 5 respectively). We have experimented with 2-dimensional orthogonal range queries over MDRS and 3-sided queries over 3SIDED and MDRS (we regard a 3-sided query as a range query with $d = \infty$ or $d = -\infty$, where d is the unbounded side). Initially we set the test beds to conduct our experiments, i.e. the number of queries and peers, the datasets, etc. Finally we evaluate each protocol by plotting its results based on the cost model presented on Section 6.2.

6.3.1 Datasets

We have simulated MDRS and 3SIDED with five datasets; one realistic and three synthetic ones. Two of the synthetic were tuff to handle efficiently and one we pretty easy. All of them are 2-D and contain about 1M data points. They are all listed on Figure 6.3.1. The realistic dataset is depicted on Figure 6.1(d) and illustrates roads on he map of Greece. The easy synthetic dataset contains data points following a random uniform distribution (see Figure 6.1(e)). Lastly, the remaining tuff synthetic datasets contain data points generated from the following distribution: (a) a circle distribution (Figure 6.1(a)), (b) 25 Gaussian clusters (Figure 6.1(b)) and (c) a diagonal distribution (Figure 6.1(c)).

6.3.2 Queries

For the MDRS we have crafted pseudo 3-sided queries bounded on top, i.e. 2-D range queries shifted on the x-axis (i.e. $y_{min} = 0$). For the 3SIDED we have created reduced 3-sided queries bounded on top as described on the following paragraph. Both type of queries have been created data skewed, i.e. the probability for a key to answer a query is analogous to the key density around the location of this key. Therefore peers with many data are very popular and expected to accept many queries.

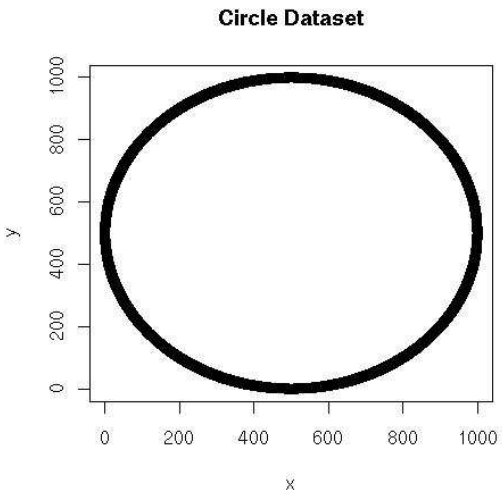
Creation of 3-sided queries

In order to create 2-D range queries shifted on the x-axis we have followed the following steps (assume that we want the query to contain $qsize$ keys for answer):

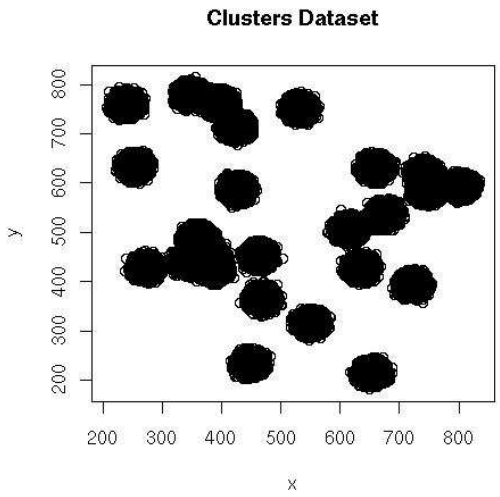
1. Choose a random point (x, y) (under the condition that the number of all the points under the horizontal line passing through point is greater or equal to $qsize$ else reinitialize). The chosen point is at this point a rectangular that closes in zero volume.
2. Extend a little bit the rectangular from the left and the right direct ions on the x-dimension.
3. Extend a little bit the rectangular from the top and the bottom direct ions on the y-dimension.
4. If the rectangular includes approximately d (with a predefined positive/negative bias) data then stop. Else repeat the previous two steps iteratively until convergence.

Creation of 4-sided queries

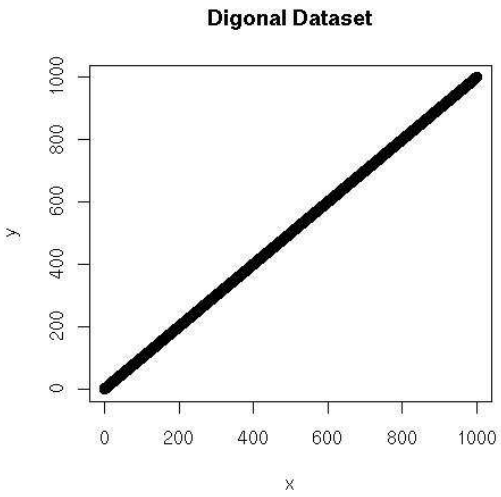
In order to create data skewed 3-sided queries we have followed the following steps (assume that we want the query to contain $qsize$ keys for answer):



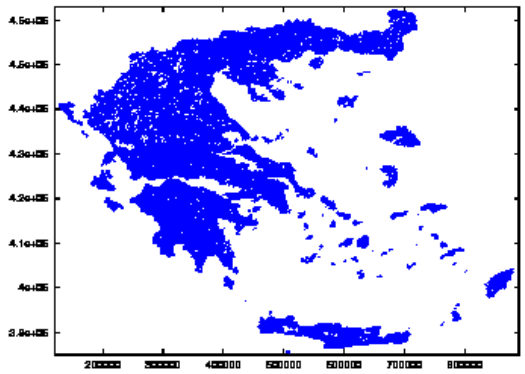
(a) Circle



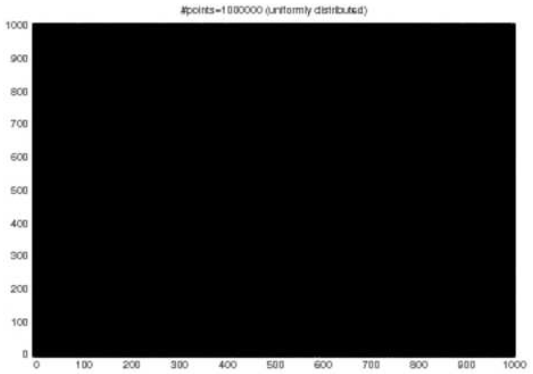
(b) Clusters



(c) Diagonal



(d) Greece



(e) Uniform

Figure 6.1: Available Datasets with which we have experimented.

1. Choose a random point (x, y) (under the condition that the number of all the points under the horizontal line passing through point is greater or equal to $qsize$ else reinitialize).
2. Extend a little bit the point from the left and the right directions thus creating a line segment on the x-dimension.
3. If the rectangular area under the line segment includes approximately d (with a predefined positive/negative bias) data then stop. Else repeat the previous two steps iteratively until convergence.

6.3.3 Bootstrapping Algorithm

The bootstrapping algorithm (see Section 3.3) that was followed by both protocols for the construction of the trie was the Data Balanced Selection. Therefore the nodes should tend to have equal number of data. The validity of this assertion will be validated later when we will discuss The Fairness of Index of each protocol.

6.3.4 Test Beds

Each experiment's network is consisted of $1K$, $5K$, $10K$, $20K$, $30K$, $50K$, $75K$ and $100K$ number of peers. The number of queries is relevant to the number of peers and equal to the one third of them. For example when the number of peers is equal to $1K$ then we initiate $1K/3 = \lfloor 333 \rfloor$ queries. The datasets we have employed are the cycle, clusters, diagonal and uniform ones that are presented on Section 6.3.1.

Maximum Throughput (see Section 6.2.3) is reverse proportional to the messages the most loaded peers receives. These messages can be divided into the ones locally answered (relative messages) and the ones forwarded (non-relative messages). When the size of the query is small (i.e. the number of the reported data items that answer the query) we expect the number of the relevant messages to be low and the number of the non-relative messages to be high. On the contrary we expect the opposite when the size of the query is high. In order to further examine the relation of Maximum Throughput in respect to the query size we have create 3 query sizes, 50–60, 500–600 and 5000–6000.

In order to avoid clutterness due to the large number of figures we have grouped into one figure the three figures that correspond to an experiment of the same specs except for the three possible query sizes. In order to accomplish this we have used improperly for each experiment and each network size the notation of the *confidence intervals* instead of the notation of points. Each curve describes an experiment carried out for three query sizes and a range of peers. Each bundle of three query sizes is depicted on the trajectory of each curve with a confidence interval which contains three points, one for each query size. These three points correspond to three results sorted from the lowest to highest. Accordingly the have been depicted on the bar of each confidence interval. More specifically each c.i. contains a *mean* value, a *lowest* bound and a highest *highest* bound. The lowest bound corresponds to the experiment with the query size that gave the lowest performance value, the opposite gilts for the highest bound and the one point left corresponds to the value of the remaining experiment. Therefore a confidence interval shows the range that a performance metric gets for an experiment with three query sizes: 50–60, 500–600 and 5000–6000 points.

We have carried out 10 individual experiments for both MDRS and 3SIDED protocols. We have used different seed for the random generator for each experiment and averaged over the results.

6.3.5 Overview of the experiments conducted

The number of parameters that we have set up in order to conduct the experiments are quite a few because we want to thoroughly study the behavior of MDRS and 3SIDED. Therefore the reader may be confused. For this reason we sum up here all the test beds that we have set up:

For 2-dimensional orthogonal range queries over MDRS we have:

- 1 protocol: MDRS
- 5 datasets
- 3 query sizes
- 10 simulations

And therefore we run in total $1 * 5 * 3 * 10 = 150$ test beds.

For 3-sided range queries over MDRS and 3SIDED we have:

- 2 protocols:
  - MDRS
  - 3SIDED (4 possible values for Y_{SEP})
- 5 datasets
- 3 query sizes
- 10 simulations

And therefore we run in total $(1 + 4) * 5 * 3 * 10 = 750$ test beds.

6.3.6 Results: 2-dimensional Rectangular Searching Over MDRS

On the following paragraphs we experiment with 2-dimensional rectangular range queries over MDRS. The metrics used are Fairness Of Index, Replication, Latency, Maximum Throughput and Traffic.

Fairness Index

As we have already mentioned on Section 6.3.3 we expect all the peers to have comparable number of data items in their disposal because of the Data Balanced Selection choice as bootstrapping algorithm.

Initially, remember from Section 6.2.2 that FI is bounded between 0 and 1, where 1 denotes the perfect fairness where all the peers have exactly the same number of data items and the opposite happens for FI equal to 0.

This is confirmed as we case see on Figure 6.2. On x-axis we present the network size and on y-axis the FI for each network size. Actually FI is generally high, especially for the uniform dataset where it approaches 0.7 (remember from Section 6.2.2 that FI is bounded between 0 and 1, where 1 denotes the perfect fairness where all the peers have exactly the same number of data items and

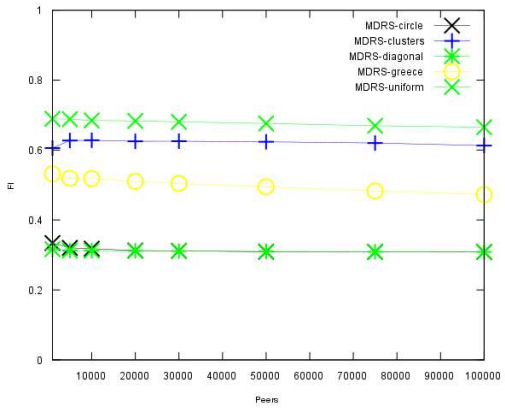


Figure 6.2: Fairness Index (for orthogonal range queries over MDRS)

the opposite happens for FI equal to 0). The worst FI is achieved by the Circle and Diagonal datasets and is equal to 0.3. The FI of the Greece dataset is lied somewhere between and is equal to 0.5.

Latency

Latency is theoretically (see Section 3.4) equal to $O(logn)$, where n is the number of peers. On Figure 6.3 we present the Latency for a series of network sizes. Obviously MDRS approaches indeed this theoretical measure. It's obvious that Latency is independent of the query size.

Average Messages Per Query

The Average Message Traffic is increased with the number of peers as showed on Figure 6.4. On x-axis we list the number of the peers of the network and on y-axis the Average Messages Per Query for each such network size. All the datasets present comparable Average Message Traffic with the best (lowest) having the Diagonal dataset and the worst (highest) having the Uniform dataset. The obvious relation between the Traffic and the query size is the following: Traffic(small query size) $\ll$ Traffic(big query size).

Maximum Throughput

Maximum Throughput in relation to the network size is depicted on Figure 6.5. On x-axis we list the number of the peers of the network and on y-axis the Average Messages Per Query for each such network size. It's obvious that MDRS presents the maximum (best) Maximum Throughput for the Diagonal dataset and the lowest (worst) for the Circle dataset. Generally Maximum Throughput scales with the number of peers. The obvious relation between the Maximum Throughput and the query size is the following: Max.Throughput(small query size) $\gg$ Max.Throughput(big query size)

6.3.7 Results: 3-sided Range Searching Over MDRS and 3SIDED

On the following paragraphs we experiment with 3-sided queries over MDRS and 3SIDED. The metrics used are again Fairness Of Index, Replication, Latency, Maximum Throughput and Traffic.

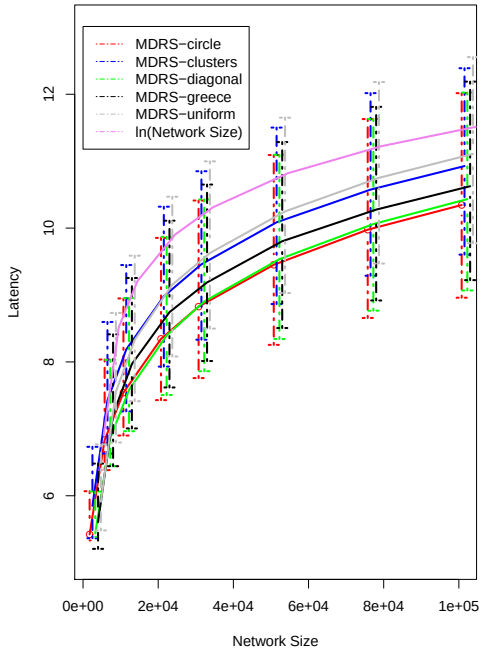


Figure 6.3: Latency (for orthogonal range queries over MDRS)

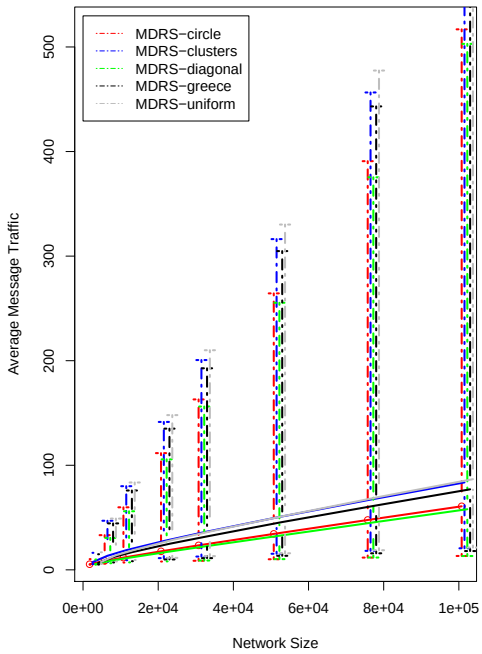


Figure 6.4: Average Message Traffic (for orthogonal range queries over MDRS)

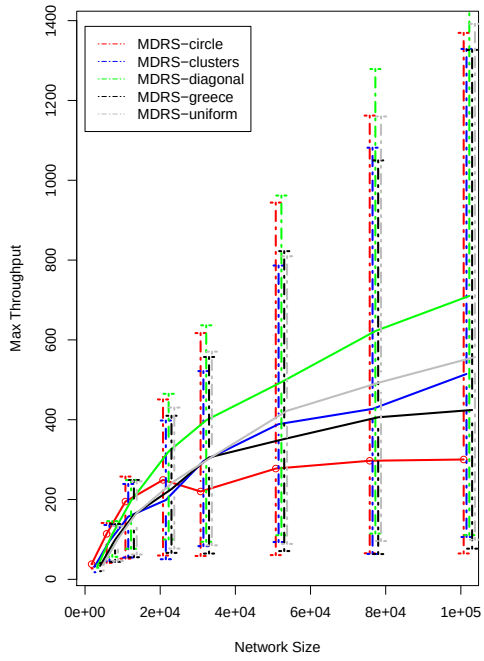


Figure 6.5: Maximum Throughput (for orthogonal range queries over MDRS)

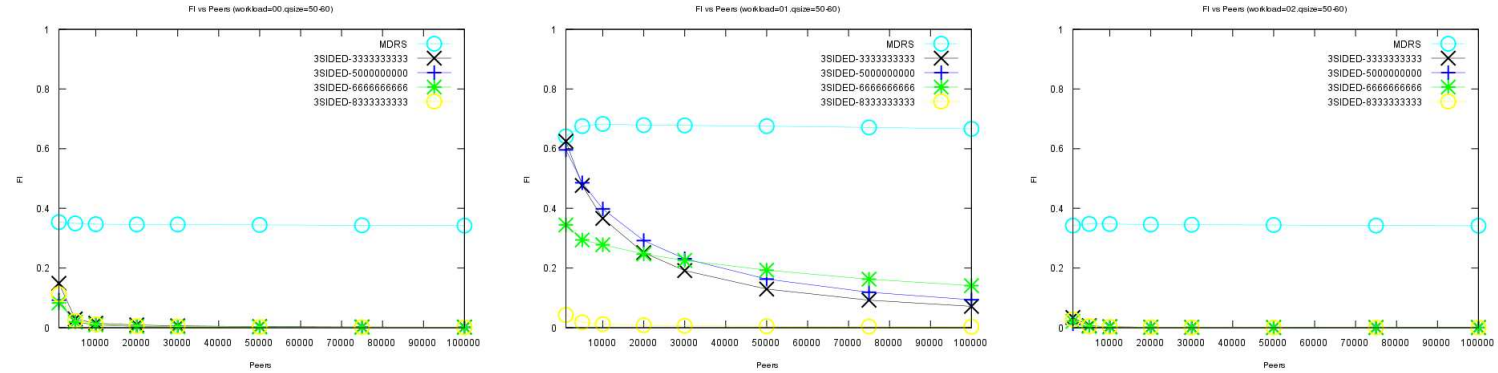
Fairness Index

Initially, remember from Section 6.2.2 that FI is bounded between 0 and 1, where 1 denotes the perfect fairness where all the peers have exactly the same number of data items and the opposite happens for FI equal to 0.

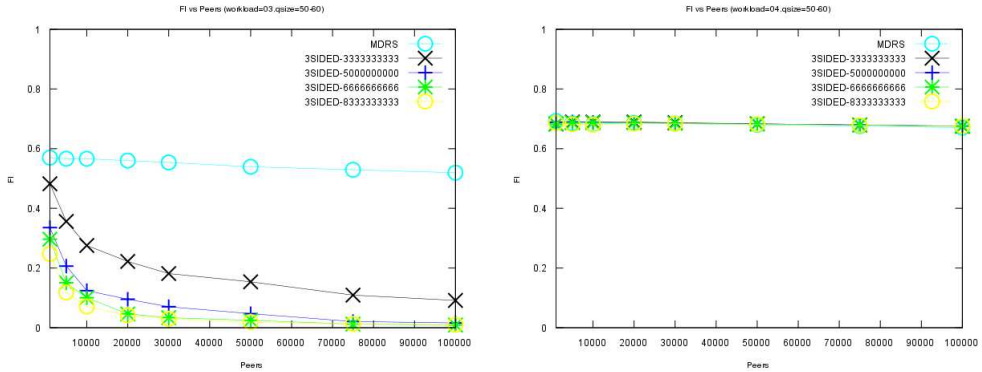
Fairness Of Index is depicted on Figure 6.3.7. On x-axis we present the network size and on y-axis the FI for each network size. As we have already mentioned on Section 6.3.3 we expect all the peers to have comparable number of data items in their key space because of the Data Balanced Selection bootstrapping algorithm selection made. Unfortunately this is the true in our case. As we see on Figure 6.3.7 FI is quite high for both MDRS and 3SIDED, especially for the uniform dataset where FI approaches 0.7. For the rest datasets the FI of MDRS is generally better (higher) and constant but for 3SIDED decreases exponentially and becomes soon less that 0.1. More specifically 3SIDED generally has Fairness Index than 0.4 for the datasets Circle,Clusters,Diagonal and Greece. MDRS behaves better with higher FI.
An importance observation is the inverse relation between 3SIDED's Y_{SEP} and FI. Namely, higher Y_{SEP} the lower FI is.

Replication

Replication is depicted as a histogram on Figure 6.7. X-axis is graduated with the datasets and y-axis measures the replication of each dataset. Obvious replication is always greater than 1.



(a) Circle (b) Clusters (c) Diagonal



(d) Greece (e) Uniform

Figure 6.6: Fairness Index (for 3-sided range queries over MDRS and 3SIDED)

Obviously MDRS has replication equal to 1 because there aren't any peers holding the same keys (see Section 4.1). Generally. Replication is less than 7.5. This means that replication is bounded. The obvious relation between the Replication and Y_{SEP} is the following: Replication(high Y_{SEP}) $\gg$ Replication(low Y_{SEP}).

An importance observation is the relation between 3SIDED's Y_{SEP} and Replication which seems to be proportional. The last result is expected since the higher Y_{SEP} is the higher on the y-axis the peers are places and therefore their key spaces are larger with more keys.

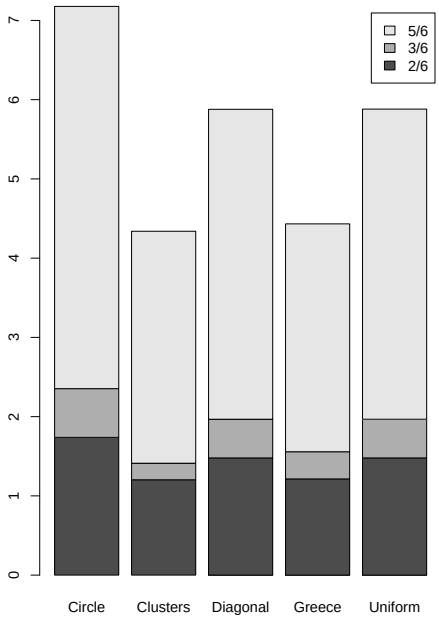
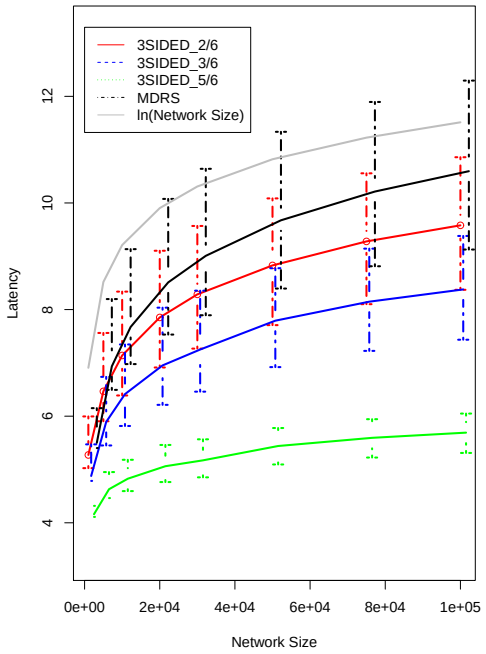


Figure 6.7: Replication for the 3SIDED protocol

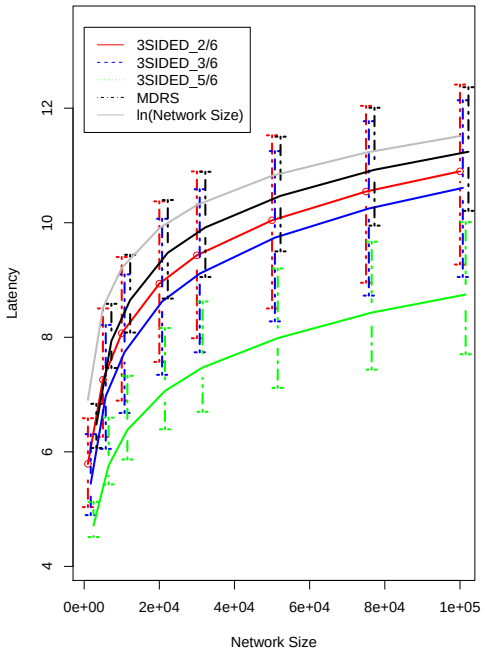
Latency

Latency is theoretically (see Section 3.4) equal to $O(logn)$, where n is the number of peers. On Figures 6.8 and 6.9 we depict the Latency. On x-axis we list the network size and on the y-axis the Latency for each network size. From the figures we see that both MDRS and 3SIDED approach indeed this theoretical measure. But globally 3SIDED has less Latency than MDRS (minor deviation). Note than the Latency is increasing while Y_{SEP} is increasing (for the 3SIDED protocol). This is expected because the replication is getting higher while Y_{SEP} is getting higher and therefore more peers can answer a query.

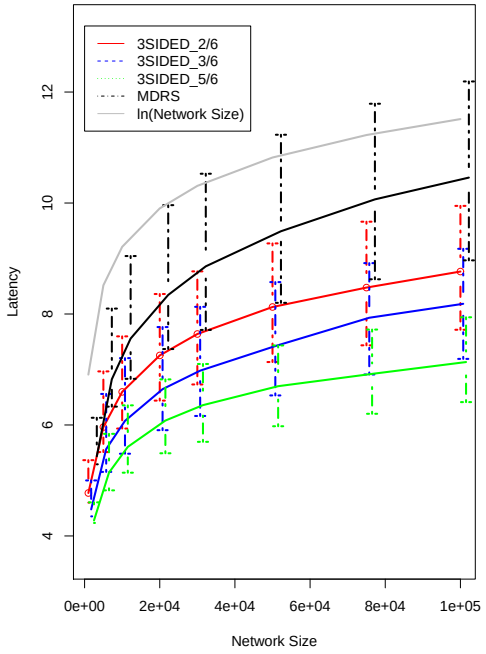
Latency is theoretically (see Section 3.4) equal to $O(logn)$, where n is the number of peers. From Figure 6.3 we see that both MDRS and 3SIDED approach this theoretical measure. It's obvious that Latency is independent of the query size. The obvious relation between the Latency and query size is the following: Latency(small query size) $\approx$ Latency(big query size) $\Rightarrow$ Latency.



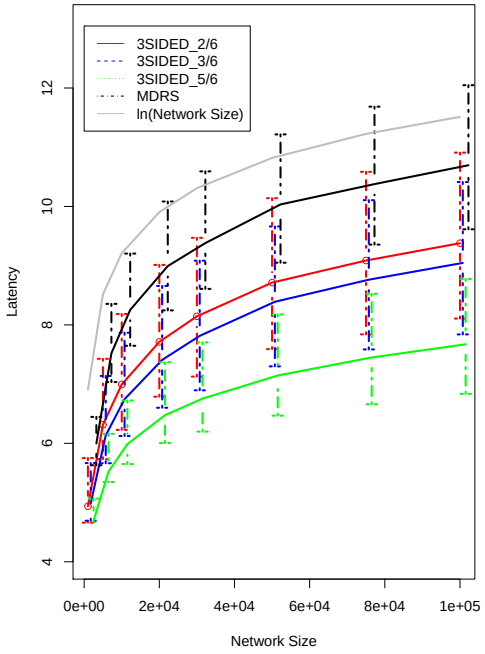
(a) Circle



(b) Clusters



(c) Diagonal



(d) Greece

Figure 6.8: Latency (for 3-sided range queries over MDRS and 3SIDED)

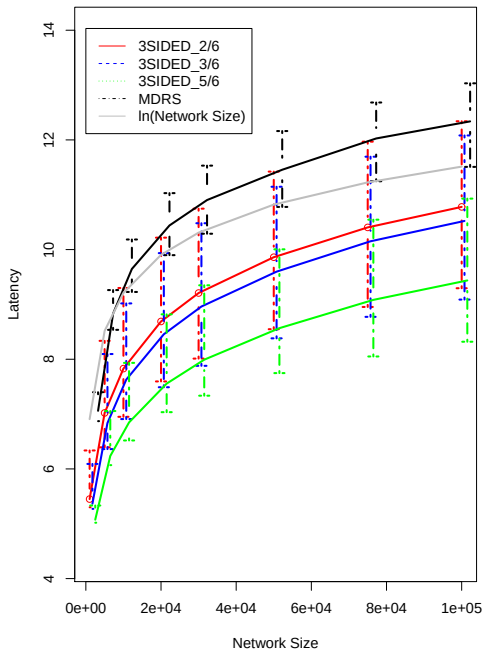


Figure 6.9: Latency (for 3-sided range queries over MDRS and 3SIDED)

Average Messages Per Query

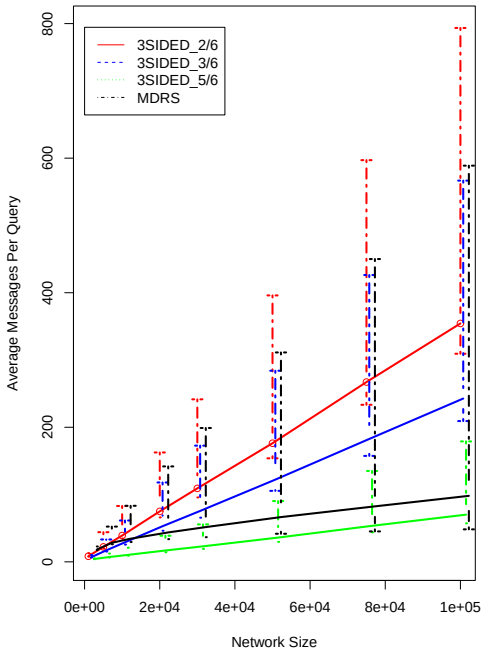
Average Message Per Query (or else Traffic) is depicted on Figures 6.10 and 6.11. On x-axis we list the number of the peers of the network and on y-axis the Average Messages Per Query for each such network size.

Obviously, MDRS grows rapidly for all y_{sep}). Both MDRS and 3SIDED for $y_{sep} = 5/6$ have good behavior. On the contrary 3SIDED for y_{sep}={2/6,3/6} has bad behavior and doesn't scale. The obvious relation between the Traffic and y_{sep} is the following: Traffic(high y_{sep}) $\ll$ Traffic(low y_{sep}). The obvious relation between the Traffic and query size is the following: Traffic(small query size) $\ll$ Traffic(big query size).

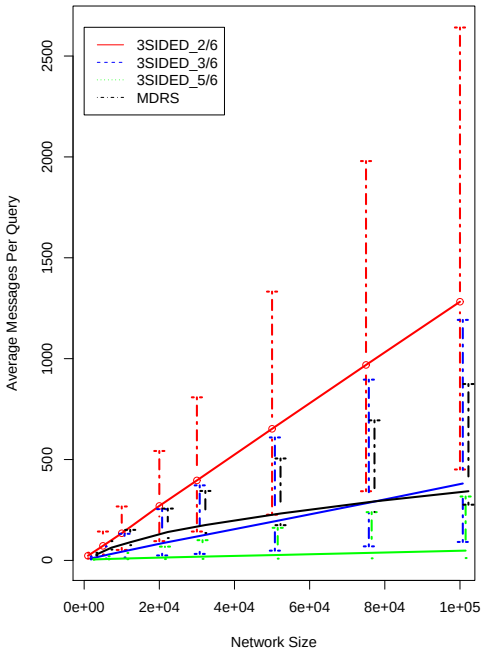
Maximum Throughput

Maximum Throughput in relation to the network size is depicted on Figures 6.12 and 6.13. On x-axis we list the number of the peers of the network and on y-axis the Average Messages Per Query for each such network size.

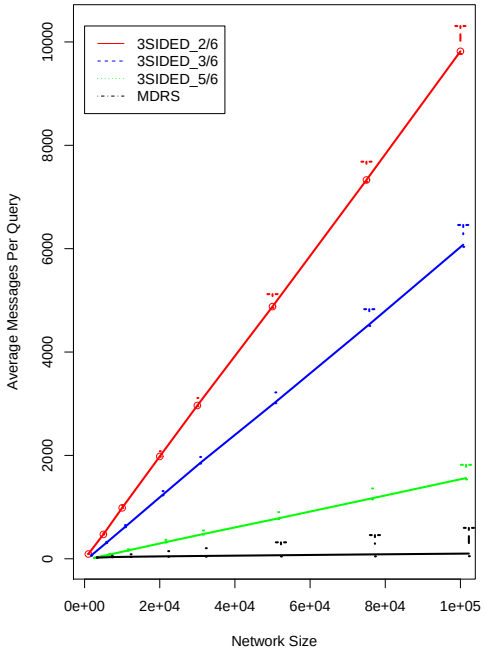
3SIDED generally has higher Maximum Throughput than MDRS. Maximum Throughput for MDRS is generally constant. Constant or decreasing Maximum Throughput means that the network isn't scalable with the number of peers, or in other words if the network size reaches a specific high size then a peer will become a major bottleneck. 3SIDED for the Uniform dataset does scale. Comparing 3SIDED and MDRS on the Uniform dataset we see that 3SIDED achieves better Maximum Throughput by a factor of 1000 versus MDRS.3SIDED for all the rest datasets and MDRS for all datasets don't scale. The obvious relation between Maximum Throughput and query size is the following: Max.Throughput(small query size) $\gg$ Max.Throughput(big query



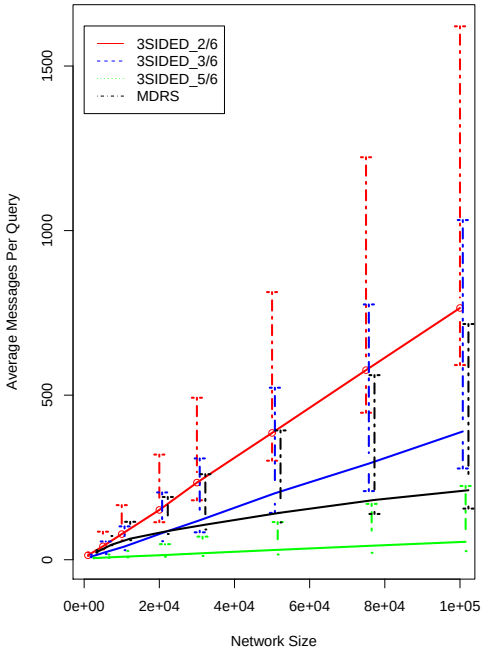
(a) Circle



(b) Clusters



(c) Diagonal



(d) Greece

Figure 6.10: Average Message Traffic (for 3-sided range queries over MDRS and 3SIDED)

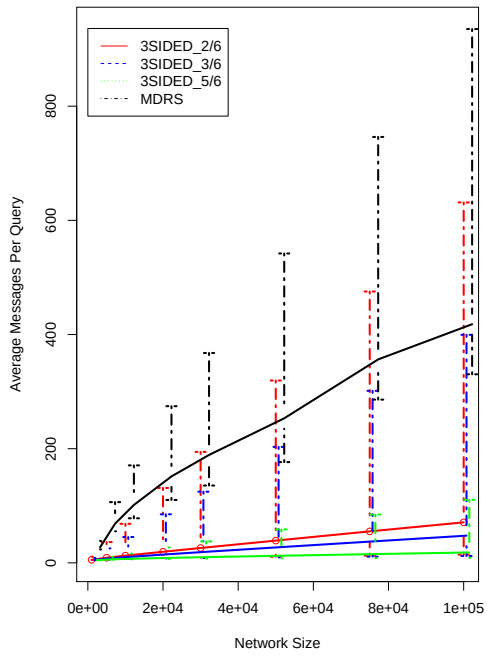
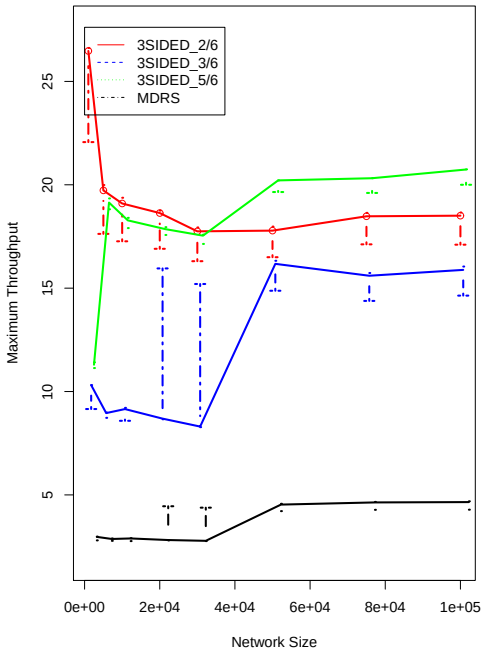
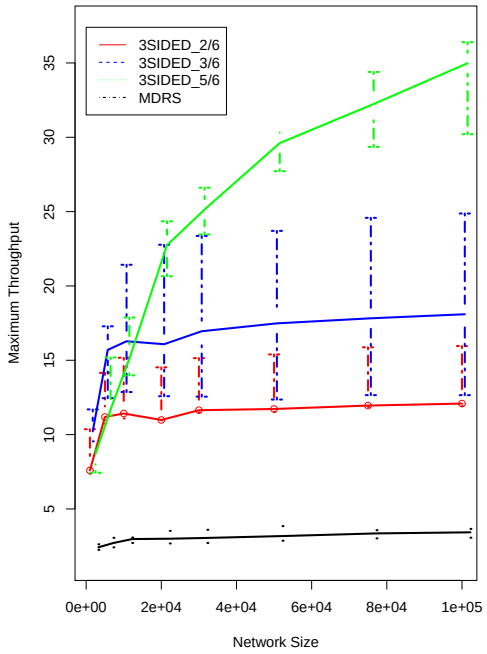


Figure 6.11: Average Message Traffic (for 3-sided range queries over MDRS and 3SIDED)

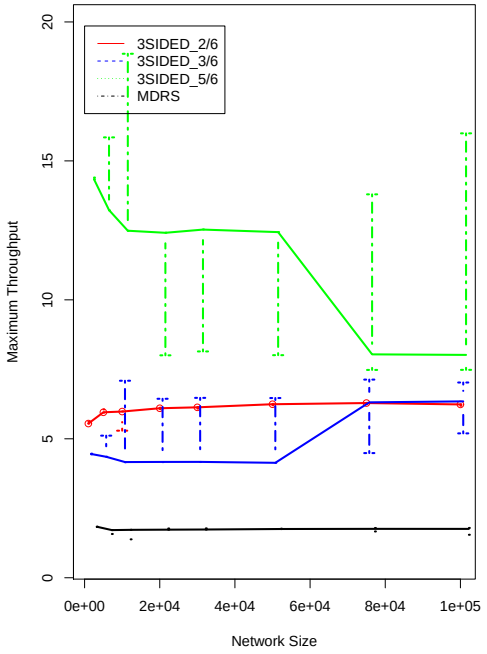
size).
An important notice is the fact that Maximum Throughput is increased when Y_{SEP} is increased as well. This is expected because the number of peers than can answer a query is increased when Y_{SEP} is increased because the replication is also increased (as we saw on Section **??**).



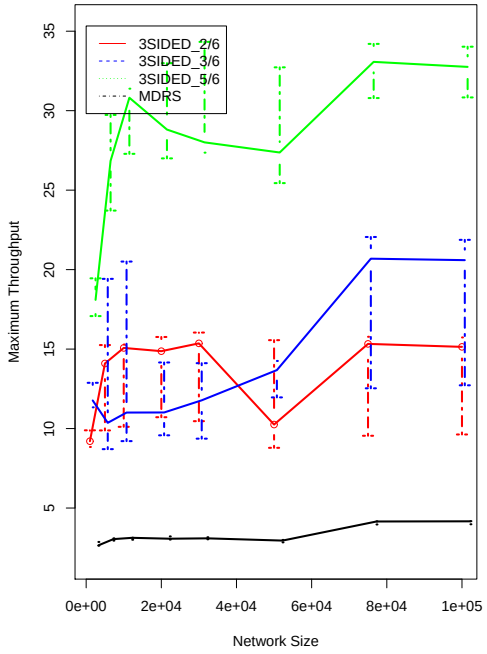
(a) Circle



(b) Clusters



(c) Diagonal



(d) Greece

Figure 6.12: Maximum Throughput (for 3-sided range queries over MDRS and 3SIDED)

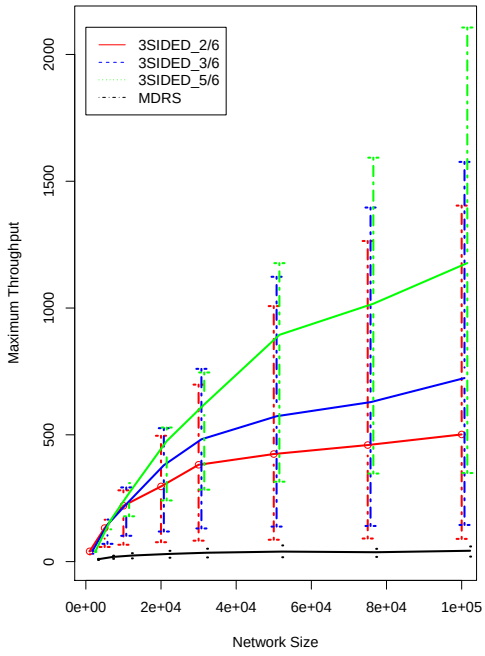


Figure 6.13: Maximum Throughput (for 3-sided range queries over MDRS and 3SIDED)

Chapter 7

Conclusion and Future Work

Throughout the previous chapters we have presented GRaSP, a framework for constructing distributed data structures for answering generalized queries. This means that one can customize GRaSP to a specific shape and dimensionality of key (e.g. point, rectangular, polygon, etc) and query (e.g. point, rectangular, polygon, etc) and rapidly construct new distributed data structures without the need of embroiling with low details such as communication details. GRaSP supports data updates (insertions/deletions), redundancy via the customization of the space partitioning. GRaSP is provable efficient in terms of latency and congestion. More specifically, the routing diameter (with high probability), the latency (i.e. the number of hops) and the congestion is logarithmic to the number of peers of the network.

We have evaluated GRaSP framework empirically by constructing two new protocols, MDRS which handles d-dimensional orthogonal range queries and 3SIDED which handles d-dimensional 3-sided queries. 3SIDED is the only protocol on our knowledge that can handle 3-sided queries on P2P. Lastly we have experimented with synthetic and realistic datasets and evaluated MDRS over planar orthogonal range queries and compared both protocols over planar 3-sided queries.

The cost model used in order to conduct those experiments measured the scalability performance of the networks in terms of latency, fairness index, average messages per query, maximum throughput and replication (where appropriate).

In the immediate future we plan to support other types of search problems such as Nearest Neighbor, similarity search, aggregate, etc. We also want to deploy the existing and the aforementioned protocols on real conditions such as on the PlanetLab. So far we have handled load balancing via static countermeasure, i.e. user-defined space partitioning techniques that exploit the nature of each problem. For example in the case of the 3SIDED protocol we have placed many peers low on the y-dimension because there there was a high probability a query to refer. We now want to craft and experiment with more general and dynamic solutions so that the network can adapt to load imbalances.

GRaSP has been published as part of the article *GRaSP: Generalized Range Search in P2P Networks*[37].

Bibliography

[1] Don batory. on the difference between very large scale reuse and large scale reuse. in larry latour, steve philbrick, and chandu bhavsar, editors, proceedings of the fourth annual workshop on software reuse, november 1991.

[2] *Chord: A scalable peer-to-peer lookup service for internet applications* (2001).

[3] *Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems* (London, UK, 2001), Springer-Verlag.

[4] *A scalable content-addressable network* (2001).

[5] *Kademlia: A Peer-to-Peer Information System Based on the XOR Metric.* (2002).

[6] *Tapestry: A resilient global-scale overlay for service deployment* (2003).

[7] *Updates in Highly Unreliable, Replicated Peer-to-Peer Systems* (2003).

[8] *One Torus to Rule them All: Multidimensional Queries in P2P Systems* (2004).

[9] *VBI-Tree: A Peer-to-Peer framework for supporting multi-dimensional indexing schemes* (2006).

[10] ABERER, K. P-Grid: A self-organizing access structure for P2P information systems. *Sixth International Conference on Cooperative Information Systems (CoopIS 2001), Lecture Notes in Computer Science 2172* (2001), 179–194.

[11] ABERER, K., PUNCEVA, M., HAUSWIRTH, M., AND SCHMIDT, R. Improving data access in p2p systems. *IEEE Internet Computing 6*, 1 (2002), 58–67.

[12] ARGE, L., EPPSTEIN, D., AND GOODRICH, M. T. Skip-webs: Efficient distributed data structures for multi-dimensional data sets. In *PODC* (2005), pp. 69–76.

[13] ASPNES, J., AND SHAH, G. Skip graphs. *ACM Transactions on Algorithms 3*, 4 (Nov. 2007), 37.

[14] BAYER, R. Binary b-trees for virtual memory. In *Proceedings of 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, San Diego, California, November 11-12, 1971* (1971), E. F. Codd and A. L. Dean, Eds., ACM, pp. 219–235.

[15] BAYER, R. The universal b-tree for multidimensional indexing: general concepts. In *WWCA '97: Proceedings of the International Conference on Worldwide Computing and Its Applications* (1997), pp. 198–209.

[16] BECKMANN, N., KRIEGEL, H.-P., SCHNEIDER, R., AND SEEGER, B. The r*-tree: an efficient and robust access method for points and rectangles. *SIGMOD Rec. 19*, 2 (1990), 322–331.

[17] BENTLEY, J. L. Multidimensional binary search trees used for associative searching. *Commun. ACM 18*, 9 (1975), 509–517.

[18] BENTLEY, J. L. Multidimensional binary search in database applications. *IEEE Trans. Software Eng. 4*, 5 (1979), 333–340.

[19] BERCHTOLD, S., KEIM, D. A., AND KRIEGEL, H.-P. The X-tree: An index structure for high-dimensional data. In *Proceedings of the 22nd International Conference on Very Large Databases* (San Francisco, U.S.A., 1996), T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, Eds., Morgan Kaufmann Publishers, pp. 28–39.

[20] CHEN, H. H., AND HUANG, T. S. A survey of construction and manipulation of octrees. *Computer Vision, Graphics, and Image Processing 43*, 3 (September 1988), 409–431.

[21] CIACCIA, P., PATELLA, M., AND ZEZULA, P. M-tree: An efficient access method for similarity search in metric spaces. In *The VLDB Journal* (1997), pp. 426–435.

[22] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.

[23] CRAINICEANU, A., LINGA, P., GEHRKE, J., AND SHANMUGASUNDARAM, J. Querying peer-to-peer networks using P-trees. In *WebDB* (2004), pp. 25–30.

[24] DATTA, A., HAUSWIRTH, M., JOHN, R., SCHMIDT, R., AND ABERER, K. Range queries in trie-structured overlays. In *P2P '05: Proceedings of the Fifth IEEE International Conference on Peer-to-Peer Computing (P2P'05)* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 57–66.

[25] DAVID A. WHITE, R. J. Similarity indexing with the sstree. *In Proceedings of the 12th ICDE Conference* (1996), 516–523.

[26] FINKEL, R. A., AND BENTLEY, J. L. Quad trees: A data structure for retrieval on composite keys. *Acta Inf. 4* (1974), 1–9.

[27] GOODRICH, M. T., NELSON, M. J., AND SUN, J. Z. The rainbow skip graph: a fault-tolerant constant-degree distributed data structure. In *SODA* (2006), pp. 384–393.

[28] GUTTMAN, A. R-trees: a dynamic index structure for spatial searching. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 1984), pp. 47–57.

[29] HARVEY, N. J. A., JONES, M. B., SAROIU, S., THEIMER, M., AND WOLMAN, A. Skipnet: a scalable overlay network with practical locality properties. In *USENIX Symp. on Internet Technologies and Systems* (2003), pp. 9–9.

[30] HELLERSTEIN, J. M., NAUGHTON, J. F., AND PFEFFER, A. Generalized search trees for database systems. In *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland* (1995), U. Dayal, P. M. D. Gray, and S. Nishio, Eds., Morgan Kaufmann, pp. 562–573.

[31] JAGADISH, H., OOI, B., AND VU, Q. Baton: A balanced tree structure for peer-to-peer networks, 2005.

[32] JAIN, R., CHIU, D., AND HAWE, W. A Quantitative Measure Of Fairness And Discrimination For Resource Allocation In Shared Computer Systems. *ArXiv Computer Science e-prints* (Sept. 1998).

[33] JELASITY, M., JESI, G. P., MONTRESOR, A., AND VOULGARIS, S. Peersim. Online webpage at http://peersim.sourceforge.net, 2006.

[34] KAMEL, I., AND FALOUTSOS, C. Hilbert r-tree: An improved r-tree using fractals. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1994), Morgan Kaufmann Publishers Inc., pp. 500–509.

[35] LOMET, D. B., AND SALZBERG, B. The hb-tree: a multiattribute indexing method with good guaranteed performance. *ACM Trans. Database Syst. 15*, 4 (1990), 625–658.

[36] MICHAEL T. GOODRICH, R. T. Algorithms design. In *Algorithms Design*. Wiley, 2002.

[37] MICHAIL ARGYRIOU, VASILIS SAMOLADAS, S. B. Grasp: Generalized range search in peer-to-peer networks. In *InfoScale* (Napoli, Italy, June 4-6 2008).

[38] RAMABHADRAN, S., RATNASAMY, S., HELLERSTEIN, J. M., AND SHENKER, S. Prefix hash tree, an indexing data structure over distributed hash tables. Tech. rep., Intel Research Berkeley, Feb. 2004.

[39] ROBINSON, J. T. The k-d-b-tree: a search structure for large multidimensional dynamic indexes. In *SIGMOD '81: Proceedings of the 1981 ACM SIGMOD international conference on Management of data* (1981), pp. 10–18.

[40] SELLIS, T., ROUSSOPOULOS, N., AND FALOUSTOS, C. The R+ -tree: A dynamic index for multi-dimensional objects. In *vldb* (Brighton, England, 1987), pp. 507–518.

[41] SITZMANN, I., AND STUCKEY, P. J. O-trees: A constraint-based index structure. In *Australasian Database Conference* (2000), pp. 127–134.

[42] SPYROS BLANAS, V. S. Contention-based performance evaluation of multidimensional range search in peer-to-peer networks. In *InfoScale* (Suzhou, China, June 6-8 2007), ACM.

[43] YANG, W. S., CHUNG, Y. D., AND KIM, M. H. The rd-tree: a structure for processing partial-max/min queries in olap. *Inf. Sci. Appl. 146*, 1-4 (2002), 137–149.

[44] ZHENG, C., SHEN, G., LI, S., AND SHENKER, S. Distributed segment tree: Support of range query and cover query over DHT. In *IPTPS* (2006).

Index