A Bottom-Up Approach to Real-Time Search in Large Networks and Clouds

MISBAH UDDIN

Doctoral Thesis
Stockholm, Sweden, 2016
Abstract

Networked systems, such as telecom networks and cloud infrastructures, generate and hold vast amounts of configuration and operational data. The goal of this work is to make all this data available through a real-time search process named network search, which will enable new real-time management solutions. The thesis contains several contributions towards engineering a network search system. Key elements of our design are a weakly structured information model that includes spatial properties, a query language that supports location- and schema-oblivious search queries, a peer-to-peer architecture, a set of echo protocols for scalable query processing, and an indexing protocol for efficient routing for spatial queries. The data against which network search is performed is maintained in local real-time databases close to the data sources. The design follows a bottom-up approach in the sense that the topology for query routing is constructed from the underlying network topology. We have built a prototype of the system on a cloud testbed and developed applications that use network search functionality. Testbed measurements suggest that it is feasible to engineer a network search system that processes queries at low latency and low overhead and that can scale to 100'000 nodes. Simulation results for spatial queries show that query processing achieves response times and incurs overhead close to an optimal protocol, and that query result remains accurate under significant churn.
Sammanfattning

Acknowledgements

I am deeply grateful to my advisor, Prof. Rolf Stadler, for his invaluable support and constant guidance during the course of this thesis. He has always been understanding and exceptionally patient when things do not turn out as expected. I consider him as a lifelong mentor. In addition, I would like to show my gratitude to Prof. Gunnar Karlsson for giving me the opportunity to be a part of LCN.

I would like to thank all collaborators who helped to materialize this thesis project. In this regard, I feel honored to be in touch with Dr. Alexander Clemm. He contributed a lot to many aspects of this thesis. In addition, he has been a great mentor during my internship in Cisco Systems, USA. Further, I am grateful to Usman Khalid, Amy Skinner, Thanakorn Sueverachai, and Kenichi Takagiwa for their contribution in different stages of prototype implementation. In addition, I thank Masanori Miyazawa and Michiaki Hayashi from KDDI R&D Lab, Japan, and Samer Salam, Cisco Systems, USA.

I would like to thank Rerngvit Yangratokke, Sylvia Kouyoumdjieva, and Fetahi Wuhib for general help and discussions of interesting topics inside and outside of research. I would like to thank all colleagues here at LCN for maintaining a motivating and friendly environment for research. I also would like to thank Connie Linell and Anna Ohlsson for their help with administrative issues.

I am happy that my wife, Fakhrunnahar Rupa, has joined me in the journey. She endured the most in difficult times while keeping me hopeful. My daughter, Ruhani Ittela, is an inspiration for me to always do better. My parents and siblings always encouraged me to push forward. My father, Nurul Alam Mantoo, is my idol and my mother, Saida Begum, is my foundation. I feel blessed to be a part of their lives. Finally, I thank my friends in Stockholm for making this tough journey enjoyable through activities and hangouts.

Misbah Uddin,
# Table of Contents

## Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>9</td>
</tr>
<tr>
<td>1.1 Background and Motivation</td>
<td>9</td>
</tr>
<tr>
<td>1.2 Problem</td>
<td>12</td>
</tr>
<tr>
<td>1.3 Approach</td>
<td>13</td>
</tr>
<tr>
<td>1.4 The Contribution of this Thesis</td>
<td>15</td>
</tr>
<tr>
<td>1.5 Publications and Awards</td>
<td>17</td>
</tr>
<tr>
<td>2 Related Research</td>
<td>19</td>
</tr>
<tr>
<td>2.1 Information Retrieval Models</td>
<td>19</td>
</tr>
<tr>
<td>2.2 Search on the World Wide Web and the Web of Things</td>
<td>21</td>
</tr>
<tr>
<td>2.3 Very Large Databases</td>
<td>23</td>
</tr>
<tr>
<td>2.4 Peer-to-peer Information Systems</td>
<td>25</td>
</tr>
<tr>
<td>2.5 Graph Databases</td>
<td>27</td>
</tr>
<tr>
<td>2.6 Bottom-up Designs of Networked Systems</td>
<td>29</td>
</tr>
<tr>
<td>2.7 Discussion</td>
<td>29</td>
</tr>
<tr>
<td>3 Summary of Original Work</td>
<td>31</td>
</tr>
<tr>
<td>4 Open Problems for Future Research</td>
<td>35</td>
</tr>
<tr>
<td>5 List of Publications in the Context of this Thesis</td>
<td>37</td>
</tr>
<tr>
<td>6 Management by Network Search</td>
<td>39</td>
</tr>
<tr>
<td>6.1 Introduction</td>
<td>39</td>
</tr>
<tr>
<td>6.2 Related Concepts</td>
<td>42</td>
</tr>
<tr>
<td>6.3 A Framework for Network Search</td>
<td>44</td>
</tr>
<tr>
<td>6.4 A Platform for Network Search</td>
<td>46</td>
</tr>
<tr>
<td>6.5 Design Space and Challenges</td>
<td>49</td>
</tr>
<tr>
<td>6.6 Discussion</td>
<td>51</td>
</tr>
<tr>
<td>7 A Query Language for Network Search</td>
<td>53</td>
</tr>
<tr>
<td>7.1 Introduction</td>
<td>53</td>
</tr>
<tr>
<td>7.2 Related Work</td>
<td>54</td>
</tr>
<tr>
<td>7.3 An Architecture for Network Search</td>
<td>56</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS

7.4 Object Model .............................................. 57
7.5 Query Language ........................................... 59
7.6 Distributed Query Processing ............................... 61
7.7 Use Cases .................................................. 63
7.8 Discussion .................................................. 65

8 Scalable Matching and Ranking for Network Search 67
8.1 Introduction ................................................. 67
8.2 Related Work ............................................... 69
8.3 A Distributed Architecture for Network Search .......... 69
8.4 A Semantic for Matching and Ranking of Network Search Queries . 71
8.5 Distributed Processing of Search Queries .................. 74
8.6 Design and Implementation of a Search Node ............. 76
8.7 Evaluation of a Network Search Prototype on a Cloud Testbed . 77
8.8 Discussion .................................................. 80

9 A Bottom-Up Design for Spatial Search in Large Networks and Clouds 83
9.1 Introduction ................................................ 83
9.2 Architecture ............................................... 86
9.3 Distributed Index .......................................... 87
9.4 The Query Processing Protocol ............................. 92
9.5 Performance Evaluation of SpatialSearch in a Static Environment . 95
9.6 Performance Evaluation of SpatialSearch in a Dynamic Environment 99
9.7 Related Work ............................................. 102
9.8 Conclusion ................................................ 104

Bibliography ...................................................... 109
Chapter 1

Introduction

1.1 Background and Motivation

Networks and networked systems keep and produce a huge volume of configuration and operational data in configuration files, device counters, flow caches, and data collection points. This data is partitioned in the sense that it is kept in various formats, needs to be accessed by different protocols, and changes at different time scales. It takes a considerable effort to identify, localize, and collect data that are relevant to tasks involving network management including fault, configuration, accounting/administration, performance, and security (FCAPS) management. Furthermore, the complexity increases if the required data is of operational nature and needs to be accessed in real-time. Therefore, only a small portion of this data is used by today’s management processes.

We address this problem by introducing a generalized search process that makes data in network devices accessible to management processes in real-time—a concept we call network search [1]. The concept of network search can be broadly characterized in three ways [2]:

- Network search can be understood as the functionality of “googling the network in real-time” for operational data. This functionality is analogous to “googling the web” for content. Like web search, it is performed through matching search queries against a vast information space and presenting search results as ranked lists. For instance, the search query “find information about the flow <src, dest, app>” would be matched against the space that includes representations of physical and logical objects of a large IP network and may return a list of devices that the flow traverses and that impose traffic policies on the flow. However, unlike web search, network search relates to real-time information that is maintained inside the network.

- Network search can be considered as a capability that treats a network as a giant database of operational and configuration data, which can be retrieved and correlated using a database-like interface.
Network search can be seen as a functionality that generalizes network monitoring, whereby the monitoring data is not explicitly specified by its location or precise structure. Rather, the data is retrieved by characterizing its contents in simple search terms.

We believe that network search has the potential to become an important technology for the management of large networks and networked systems. Firstly, we believe that network search is likely to speed up the development process of management applications, which, in turn, will result in innovative tools for tasks in network supervision and management. Secondly, we envision that network search may become an enabling technology for new classes of network control functions. We mention some examples of tools and applications in this context:

- tracing network sessions dynamically across network boundaries for quality of service assurance;
- tracking virtualized resources dynamically in a large cloud to support adaptive cloud controllers;
- analysing the root cause of critical network faults to minimize service downtime;
- performing real-time network analytics for anomaly detection;
- performing network “garbage collection” to maximize resource utilization.

Often, information associated with networked systems has spatial semantics, which is important for many management tasks. Many use cases in network management involve searches for nearby objects. For instance, an application may search for servers within a maximum round trip delay to guarantee service-level agreements, and a surveillance system may search for \( k \) sensors close to a given location to support disaster recovery efforts. These examples motivate including spatial concepts in network search and studying spatial queries in networked systems.

We believe that network search has a potential to enable a paradigm shift in designing large-scale management systems, since it stresses an information-centric view of network management. This is illustrated in Figure 1.1.

Figure 1.1(a) presents a traditional view of network management, which is realized in many management systems. The top layer of Figure 1.1(a) presents management applications for networked systems and the bottom layer shows network devices from target technologies. Information in the devices is accessed by different protocols based on the format of the information. The applications, which use network information, rely on dedicated mechanisms for information collection, which is illustrated in the middle of Figure 1.1(a). To implement the information collection mechanism, the application requires knowledge about the format of the information, its access technology, and most importantly its location.

Figure 1.1(b) provides a flexible alternative for network management which is based on network search. Here, the middle layer contains real-time databases that
maintain information from devices in the underlying layer. It also supports functions that provide uniform, location-independent access to the information in the database to management applications that reside in the layer above. Having such an interface, management applications do not need to focus on the information collection mechanism. In addition, since information access and application modules are decoupled, the modules can be reused by other applications.

Figure 1.1: Traditional network management vs. management based on network search: (a) a framework for traditional network management; (b) a framework for network management based on network search.
We envision that the network search concept will be applicable to a wide range of networked systems. It should be applicable to a range of network technologies, for instance, enterprise, backbone, and access networks. Also, it should be applicable to large ICT infrastructures in general, for instance, large data centers and cloud technologies. This calls for design and engineering of network-agnostic solutions to the extent possible.

Since network search is often performed against operational data, which can be short-lived or fast changing, we envision that network search should be performed inside the network infrastructure, to achieve scalability in terms of fast response times and low overhead costs for search queries. A network search system should have a decentralized, self-organizing architecture with multiple access points.

If leveraged properly, recent advances in academic and industrial research and developments in the field of computing, storage, and networking make network search feasible for today’s network environments. First, the cost of computing and networking resources has decreased significantly over the last decade, which makes it reasonable to allocate potentially more powerful resources for management tasks on which a network search can be realized. Second, additional processing capacity is becoming available within network elements, some of which is available for purposes other than traffic forwarding and signaling. Third, network devices are increasingly offering APIs that can be used to significantly extend their functionality. Finally, results in the area of resource discovery, in-network aggregation, distributed real-time monitoring, very large databases, and scalable storage—often using distributed algorithms—, can be drawn upon to develop scalable search functions.

Most system designs in the above mentioned fields follow a top-down approach, which is suitable for efficient search in a large collection of information that can be easily moved from its source. However, in our case, the data against which search will be performed cannot be moved in a feasible manner from the data sources, due to its dynamic nature. We, therefore, advocate a bottom-up design of the network search system.

1.2 Problem

The key problem that we address in this thesis is that of designing and engineering a network search system. We map out the problem space by identifying a set of requirements for a network search system:

1. **Information model for operational and configuration data**: The design of a network search system must include an information model that can represent in a uniform way device data of various formats and obtained by different access methods, e.g., SNMP MIB data [3], Netflow information [4], etc. In addition, the model should support a naming scheme for uniquely identifying data objects. Also, names should be expressive so that objects can be searched using naming information.

2. **Spatial information**: The design of a network search system should support a spatial model that allows to represent object locations and distances between
1.3. Approach

We address the above design and engineering requirements using the following methods and techniques:

1. **Information model**: The challenge of finding a suitable information model is less than that of devising a new model than that of choosing one from a collection of proposed models from the fields of information retrieval, databases, and network management. For network search, a suitable option is to model device data as objects in form of attribute-value pairs. This approach allows heterogeneous data to be uniformly represented in a straightforward manner, and it allows the inclusion of data structured according to well-known information models, such as SMIv2 [5], Yang [6], and the relational model [7].

2. **Spatial model**: We choose the Euclidean model in \( \mathbb{R}^n \) to represent object locations and distances between them. The Euclidean model is simple and has been widely adopted in different contexts, including geographic information systems [8], sensor networks [9], and network-aware overlays [10, 11, 12]. Some of these systems use geographic coordinates, while others use so-called network
coordinates, which are virtual. In a network coordinate system, the Euclidean distance between two locations represents the end-to-end delay between them. We develop query processing functions for spatial queries through adopting concepts from geographic information systems.

3. **Query language:** To develop a query language for network search, we study languages for information retrieval, web search, and databases. Key language constructs should express attribute names and attribute values, since operational data can be expressed well using attribute-value pairs. Invoking a query should not require information about the structure of the object to be searched for, neither should it require the address information where the object is maintained. For query processing, we investigate the applicability of concepts that have been developed for information retrieval and web search. In particular, we devise parameterized matching and ranking functions that support different query semantics.

4. **Efficient query processing:** First, queries are processed in a distributed fashion, on local databases of objects maintained in search nodes (see Figure 1.2), which reduces query processing times. Second, on each search node a local index is maintained for fast data access. The local index is based upon the information retrieval concept of an inverted index. Third, a global index is dynamically constructed for efficient query routing and overhead reduction. The global index is a variant of an index structure used in geographic information systems. To study the tradeoffs between the above metrics, we evaluate our design choices through prototype experimentation and simulation.

5. **Scalability:** To achieve scalability of the search system in terms of the above metrics, the network search system follows a peer-to-peer design and is structured as a network of search nodes (see Figure 1.2). Further, protocols that perform key functions of the search system, including constructing the global index and query processing, are based on efficient, in-network aggregation protocols [13].

6. **Bottom-up design:** All designs of search systems we found in the literature, including web search systems, very large database systems, and spatial databases, follow a *top-down approach*. They all address the problem of efficiently processing queries on a very large database of data items that change slowly. A key aspect of efficiency is the optimal allocation of parts of this database to servers in a large infrastructure. In contrast, our design tackles the problem of efficiently processing search queries against fast changing operational data on a large number of distributed devices. In our case, the data cannot be allocated to a device, as its location is defined by the data source. For this reason, our design is *bottom-up*. 
1.4 The Contribution of this Thesis

This thesis work makes several contributions towards designing and engineering a network search system that fulfills the requirements mentioned in Section 1.2.

1. **Information model**: In our design, data in a network system is represented as objects that are modeled as a set of attribute-value pairs. Objects have a type, expressed through a specific attribute, and two objects can be linked through a joint attribute. Objects can be named using URIs or URLs, which are unique and have expressive identifiers [14]. Our information model is simple and allows to uniformly represent device data and statistics from various sources. See Chapter 7 for details.

2. **Spatial model**: We use the Euclidean space $\mathbb{R}^n$ to represent object locations. An object location is encoded in an object attribute. We developed a distributed indexing scheme to support spatial queries, which retrieve objects with specified spatial properties. We developed a protocol based on the R-tree concept (MBRIndex) that creates and maintains this distributed index and we developed an Echo protocol (SpatialSearch) that supports the execution of a range of spatial queries, including range and k-nearest neighbor queries. See Chapter 9 for more details.

3. **Query language**: We developed a query language for network search, which supports keyword search, in a similar way as web search does, through
unstructured queries that require little meta information and are useful for exploration purposes. In addition, the language supports structured queries based on relational algebra to search for objects with specific operational states, for instance. Further, the language includes operators for discovering linked objects, filtering search results, and aggregating attributes of result objects.

The language semantics allows for a range of matching and ranking options to support the specific requirements of different management applications. To achieve this, we make use of the extended boolean model (originally developed in information retrieval) [15]. See Chapter 8 for details.

4. **Bottom-up design of a peer-to-peer search system:** To support real-time search on operational data and to keep the communication overhead low, our design includes real-time databases of network objects close to the sources of operational data. This is illustrated in Figure 1.2, which shows the architecture of a network search system. The key part of this architecture is the *search plane*, in which peer-to-peer query processing is performed. The plane consists of a network of *search nodes*, which maintain the databases and run the indexing and query processing protocols.

The graph topology of the search plane for query routing, as well as the distributed index for query processing, are constructed in a bottom-up fashion. The graph topology follows the link-layer or routing topology of the underlying networked system. The distributed index is created from information in the search node databases and the graph topology. For details, consult Chapter 6 and Chapter 9.

5. **Performance of query processing and scalability:** We implemented a prototype of a network search system on our cloud testbed. The prototype design is bottom up and follows the architecture in Figure 1.2. It implements our information model, real-time object databases, query language support, and distributed query processing. Measurements on a cluster of nine high-performance servers show that the prototype can process a load of 50 queries per second with a median query latency of 40 milliseconds and a CPU overhead of below 1.5 percent. The real-time databases contain some 5000 objects on each server. From the testbed measurements and the properties of the query processing protocol, we estimate that for a cluster of 100'000 servers, for the same query load, the CPU overhead per server would stay at below 1.5 percent, while the median latency would increase to approximately 500 milliseconds. For details, see Chapter 8.

The parts of the design that relate to spatial aspects have not been implemented in the prototype. Aspects of processing spatial queries on the search plane have been evaluated through simulation. The simulation results show that our query processing protocol incurs significantly lower (up to 80%) overhead than a baseline protocol that searches all nodes. Second, the accuracy of query processing remains high for scenarios with dynamic object locations. Third, simulation results indicate that the query processing protocol becomes
more efficient with increasing network size, when compared to a baseline protocol. We performed simulation studies with network sizes beyond 100'000 nodes. Fourth, by design, the execution time of the query processing protocol is asymptotically optimal. Finally, for scenarios with dynamic object locations, there is a tradeoff between the accuracy of the search results and the rate of repair of the local index, which is controlled by the maximum message rate per link. The simulation results confirm our intuition that the higher the rate of repair, the better the accuracy of the search results and vice versa. More details can be found in Chapter 9.

6. A demonstrator for network search: We have built a fully functional demonstrator for a cloud platform. The demonstrator can execute structured and unstructured queries, whose matching and ranking semantics can be easily changed. We learned that simple management applications that use the query interface for data collection can be built fast and with small effort. (As mentioned above, there is currently no implementation of spatial search.) See Chapter 8 for details.

1.5 Publications and Awards

The results of this research have been documented in eight papers. Six of these papers have been published in peer-reviewed conferences of the network management research community, namely, IEEE/IFIP NOMS, IEEE/IFIP IM, and IEEE/IFIP CNSM. The other two papers have been submitted for publication. The papers are listed in Chapter 5.

At IETF-91, we received the Applied Network Research Prize (ANRP) has been our paper titled “Scalable Matching and Ranking for Network Search,” which is published in the proceedings of CNSM 2013.
Chapter 2

Related Research

Network search draws from concepts in information retrieval, web search, very large databases, peer-to-peer information systems, and graph databases. In this chapter, we discuss concepts in these fields that are relevant to network search.

2.1 Information Retrieval Models

Information retrieval (IR) deals with obtaining information objects (called documents in IR) from a collection of documents that are relevant to a given query [16]. A document often contains unstructured data that describes one or more entities or topics. A query, which formally states an information need, can be both unstructured, e.g., a sentence, or semi-structured, e.g., a boolean expression. A document collection can be modeled as a database of unstructured data.

In most cases, a query does not uniquely identify a single document, but matches several documents in a collection. An IR system usually computes a score of each document for a given query. The score numerically indicates how well each document in the collection matches the query and is used to rank the documents.

An IR system includes a model for representing documents, a language for generating queries, and a function that integrates matching and ranking semantics for processing queries. There are various approaches for designing an IR system. Many are based on boolean, vector space, extended boolean, probabilistic, or semantic/natural language models. We briefly discuss the first three.

The (conventional) boolean information models are based on boolean logic and set theory, whereby a document is represented as a set of terms and a query is formulated as a list of terms with boolean operators [17]. If the query matches the document description, the document is included in the query result, otherwise, it is not included. A drawback of such an approach is that it can match either too few or too many documents. In addition, all non-matched documents are treated equally. For instance, a document containing all but one query term does not match, like a document containing no query term at all. Similarly, all matched documents are treated equally, and there is no concept of ranking.

The vector space model was first introduced by Salton et al [18]. In a vector space model both a document and a query are represented as a vector of terms, where
each term is associated with a weight. The weight of a term in a document (or a query) indicates the importance of the term as a descriptor of the given document. A term that is not in a document has a weight of zero with respect to the document, and, otherwise, the weight is nonzero. The matching and ranking score depends on the cosine of the angle between the query and the document vector.

The weight of the term of a document is often based on two metrics, the term frequency \((tf)\) and the inverse document frequency \((idf)\). \(tf\) is the number of occurrences of the term within the document, and \(idf\) measures the number of documents in the collection that carry the term. Hence, \(tf\) is a document-specific metric that varies from one document to another and measures the importance of a given term within the document. On the other hand, \(idf\) is a collection-wide metric that indicates how likely a given term occurs in a randomly picked document. The weight of the term is computed as a product of the \(tf\) and the \(idf\) metrics. Therefore, a term has a large weight when it occurs many times in the document, but is rare in other documents.

Latent semantic indexing (LSI) is a well-known variant of the vector space model, which extends the model by reducing its dimensionality [19, 20]. It groups terms with similar meaning using singular value decomposition. Although LSI is known to improve the diversity of the results for a given query, compared to the basic vector model, it comes at the increased cost of processing and storage overheads.

Extended boolean models have been developed to overcome some drawbacks of the classical boolean models [21]. In contrast to a query in a classical boolean model, processing a query in an extended boolean model produces a number in the range of 0-1, which corresponds to the degree to which the query matches the document. Extended boolean models also make use of the weights assigned to the query terms, which allows certain terms to influence the ranking more than others.

Out of all extended boolean models, the p-norm model has the most desirable properties in the sense that the model tends to evaluate queries best with respect to a user’s judgement [15, 22]. This is because the p-norm model provides a tunable matching scheme, which allows for realizing a range of matching semantics, from exact matching, similar to conventional boolean models, to very loose, approximate matching, similar to vector space models.

IR has been the key technology in the early search systems for the World Wide Web, e.g., Altavista, Excite, and Infoseek, which were known as full-text search engines. These search engines presented web users with a keyword-search interface supported by indexes of web page texts and ranking mechanisms building on those presented above. However, the engines adapted information retrieval technologies to indexes containing tens of millions of documents, which were much larger than any prior information retrieval system in the public domain.

Information retrieval (IR) provides concepts on how to model semi-structured data with heterogeneous structures. All IR systems support some sort of keyword-based queries, which form the foundation for network search queries. The field of IR has also developed the concepts of matching and ranking with respect to queries, which we believe to be essential for network search. Specifically, the conventional boolean model provides an exact matching semantics for keyword queries with boolean operators, which is useful for applications of network search that re-
quire database-style matching and no ranking. The vector space model, on the other hand, offers a loose, approximate matching semantics, which is suited for applications that explore the wide space of data in a network system. The extended boolean model enables parameterization to support a range of matching options through the same search interface.

2.2 Search on the World Wide Web and the Web of Things

As discussed in Section 2.1, web search can be seen as an application of IR models to the domain of the World Wide Web. Web search is usually performed by matching keywords against static content in web pages of the World Wide Web [16]. A web search engine maintains a distributed database of inverted indexes of web pages in a dedicated infrastructure outside the web. Matching is performed on this index database, which is populated by so-called crawlers, which continuously navigate the web following hyperlinks, i.e., the link between web pages. Search results that contain matched web pages are presented as ranked lists of links to the pages.

The ranking of a web page in a traditional web search engine is determined by its keyword relevance score, which is computed using IR models, such as Okapi BM25 [23] and statistical language models [24, 25]. These models have been further extended to field-based IR models, such as BM25F [23] and “a mixture of language” models [26], which partition the information on a web page into different fields and match keywords from queries against these fields. In addition to keyword scores, ranking schemes in web search consider scores produced by hyperlink analysis algorithms, such as PageRank [27] and HITS [28], which measure the centrality of a web page.

Current web search engines, which search both static and dynamic content, use hundreds of metrics to compute ranking scores; in addition to PageRank, these metrics include a range of page usage statistics, matching scores from different IR models, freshness of data, etc. [29]. The weights of these metrics are generally determined using machine learning techniques applied to query logs [30, 31, 32].

With the evolution of Web 2.0, web search engines have evolved to search news, blogs, forums, microblogs, and the like [33]. Content in these new media forms expire at rates faster than the content of traditional web pages. New technologies have been developed to populate search indexes for these content forms, including partial indexing [34], RSS feeds [35] [36], and adaptive periodic crawling [37]. In addition to the metrics mentioned above, the freshness of the data is a determining factor in the ranking of search results [36, 38, 34]. Timestamps in the content are often used to determine the freshness of the content. In case they are missing, other metrics, such as crawling and indexing timestamps, are used. For each piece of content, a confidence score is computed off-line, taking into account the ratings of the content source, its author, as well as the possible presence of rich media, e.g., image, video, etc., in the content.

A recent focus of research has been search in web-based social networks, which supports concepts like friends and groups, and allows actions like sharing, liking,
CHAPTER 2. RELATED RESEARCH

recommending, etc., also known as social tagging [39]. The concept of web search has been extended with the advent of social tagging systems. In addition to web pages, social web entities, such as people, can be searched, and they can be ranked using scores generated by extensions of link analysis algorithms that consider tags between the entities, in addition to hyperlinks. Search in this context is guided by two principles. First, it relates to the user who invokes the query, since ranking of search results are influenced by the tagging of a user’s social network [40, 41]. Second, it uses social graphs as a means to guide the search process in a vast search space [42, 43].

Over the last years, research in the area of the Internet of Things, which investigates the internetworking of physical entities, gave rise to an effort called Web of Things. Entities in the Web of Things are realized as web pages, which means that they can be identified through URLs [44] and accessed through web protocols. Further, these entities span a wide range of real-world objects, from rooms in buildings to mobile phones. These entities contain sensors and thus have associated quantities, from occupancy levels to temperature readings, which are published on the web.

Search systems for Web of Things can be seen as an adaptation of web search to the Internet of Things. Such a system for Web of Things is generally realized as a hierarchy of processing nodes, which maintain distributed search indexes of sensor descriptions and associated measured quantities [45]. While it is possible to use traditional web search systems to search the Web of Things, additional matching schemes have been developed. The search systems for the Web of Things include comparing measured quantities against specific values, allowing to search, for instance, for an empty room in a building or the last known location of a cell phone. Specialized ranking schemes have been devised, which, for instance, use the probability of the values of attributes of real-world objects to rank search results [46]. Various implementations in the context of Web of Things are presented in [47, 48, 49, 50, 46].

The Web of Data has evolved recently as a way of exposing structured data on the web due to the proliferation of knowledge bases, e.g., DBPedia, Freebase, etc., and metadata embedded inside web pages [51]. Web of Data uses the principles of linked data [52], which primarily focuses on publishing information about entities and their relationships, by leveraging standards, such as the Resource Description Framework (RDF) [53] and using URIs that can be dereferenced [14].

A new breed of search systems has emerged to support keyword search in the Web of Data. These search systems provide search results in the form of links to the RDF representations of matched entities, as well as links to related entities. Many of these search engines, such as the ones presented in [54, 55, 56], represent data expressed in the Entity-Attribute-Value (EAV) model [57] and perform matching of keywords against indexes of EAV documents. For ranking, the search systems usually exploit field-based IR models, such as BM25F [23] and “mixture of language” models [26]. However, other search engines make use of graph-based models, where entities are represented as vertices and relationships between entities as edges [58, 59, 60]. These search engines match keywords against graph vertices and exploit edge structures to rank search results.
The development of network search is influenced in two ways from results in search in the World Wide Web and the Web of Things. The first relates to matching and ranking, whereby these fields use additional metrics than the known IR metrics, which turn out to be good candidates for network search. For instance, PageRank and other hyperlink analysis algorithms provide a way to rank high well-connected objects, even if they have a low keyword-relevance score. Such an approach can be applied, for instance, to find resources that point to relevant objects. Search for blogs and other live web content ranks search results according to content freshness, which can be effective in searching volatile data in a networked system. Search in social tagging systems provides several approaches to constrain the search process and navigate in a reduced search space, which can be applied to reduce the overhead of network search. The second topic area network search can learn from web search and search in the Web of Things relates to the immense information space in which the search is conducted. These fields have developed distributed approaches to construct and maintain very large index structures for an ever-changing information space. These approaches are important to study for the development of distributed query processing in network search.

2.3 Very Large Databases

Database management systems have been integral to the development of large-scale information systems. Recently, there has been a proliferation of very large databases that contain high numbers of database tuples and run on distributed systems. The design of such a database system is governed by the CAP theorem, which states that a distributed system can only guarantee, at the same time, two of the three properties: consistency, availability, and partition tolerance [61].

Distributed applications are increasingly using so-called NoSQL databases [62], primarily for two reasons. Compared to relational databases, NoSQL databases provide the flexibility to support data of dynamic structure, and they exhibit better performance in large-scale query processing [63, 64]. NoSQL database systems partition the data to achieve load distribution. Data items are replicated to guarantee better availability and response times. Many NoSQL databases provide only eventual consistency, which ensures that all replicas will eventually have the value of the last writes [65].

Two platforms that pioneered the wave of NoSQL databases are Amazon’s Dynamo [66] and Google’s BigTable [67]. Both are proprietary implementations, but were influential for the design of the later open-source systems. Two well known implementations based on the above platforms are Apache/Facebook’s Cassandra [68] and LinkedIn’s Voldemort [69].

Dynamo is a decentralized database system built for latency-sensitive applications that require most read and write operations to be performed within a few hundred milliseconds. Dynamo stores information objects as key-value pairs, which are suitable for storage and retrieval of small objects of different structure. Dynamo provides a simple query interface with a single operation, $\text{get}(key)$, which returns the value associated to $key$ from a Dynamo data store. Servers in a Dynamo system
are organized into a ring and have identical capabilities, which include data storage, retrieval, partitioning, and replication. For partitioning, the platform relies on consistent hashing to distribute the objects across the servers [70]. To achieve data consistency, Dynamo uses a protocol similar to quorum-based voting [71]. Dynamo powers Amazon web services, such as Amazon S3 and DynamoDB.

BigTable is a high-performance storage system used extensively in Google projects including web indexing, Google Maps, Google Earth, Youtube, Gmail, and personalized search. It is designed to store and retrieve large files of different structures. BigTable organizes information about a file as a row in a table, which is a sparse, distributed, and multidimensional sorted map. BigTable provides a query interface that allows rows to be retrieved using row-keys. Rows, sorted using their keys, are grouped into tablets, which become the units of distribution. Tablets are distributed by a master server that is responsible for data distribution and replication. A master server also keeps track of servers that store tablets and their replica. The master server is backed up by a second machine. For consistency, BigTable maintains row-level consistency using multiple techniques, which include serialization of parallel transactions, distributed locks, and logs of row updates. Spanner is a distributed relational database that uses BigTable as the underlying technology [72].

The above NoSQL databases, Dynamo and BigTable, are often classified as key-value stores. Another type of NoSQL database is the so-called “document store” [73]. A document store manages “documents”, whereby a document is defined as a set of attribute-value pairs. The attributes of a document are dynamically defined at runtime. The values associated to an attribute can be a scalar, a list of values, or even a set of attribute-value pairs. Apart from the data model, a key difference between key-value stores and document stores is that document stores allow more expressive queries, which can return both attribute-value pairs as well as documents. MongoDB [74] and CouchDB [75] are two examples of document store NoSQL databases. Many NoSQL platforms like the ones above achieve scalability of query response times primarily through allowing only eventual consistency. For this reason, as reported in [66], a production run of 30 days on the Amazon S3 cloud storage, which is based on Dynamo, is able to achieve around 200 milliseconds for the 99.9 percentile latency of read operations.

In the context of very large databases, many flexible information models have been developed, which can be used as starting points for a network search information model. Further, many database systems support a rich library of operators, including projection and aggregation, which are candidate operators for a network search system. Lastly, by relaxing consistency requirements, large databases can achieve low-latency query processing over a very large distributed dataset. For network search, the tradeoff between fast query processing and a high-level of data consistency is approached in the same way, namely, relaxed consistency seems acceptable in order to realize real-time search.
2.4 Peer-to-peer Information Systems

Frameworks for Structured Queries

Frameworks for structured queries are based on peer-to-peer designs to build structured database systems which generally support relational interfaces. These systems rely on so-called key-value stores, which are implemented using distributed Hash Tables (DHTs). Popular DHT systems are Content Addressable Network (CAN) [76], Chord [77], Pastry [78], and Tapestry [79]. These systems use distributed algorithms for mapping keys to a set of networked nodes that store pointers to the values, for routing queries, and for building routing tables. A survey of such DHT systems can be found in [80].

Many frameworks for processing search queries are based on the CAN architecture. In CAN, peer nodes are organized into a $d$-dimensional torus, which defines the space for the keys. Each peer is responsible for all locations in a hyper-rectangle within this torus, and it maintains a routing table with links to all peers responsible for adjacent hyper-rectangles. These hyper-rectangles are non-overlapping and fill out the whole key space. A key lookup is realized through successive forwarding of the lookup query to a peer in the routing table, until it reaches the node that stores the value for the particular key. The latency for a key lookup increases with $O(d \sqrt{n})$ for a network of $n$ peer nodes.

PIER is a well-known framework for processing structured queries [81]. It is based on CAN. Queries in PIER are processed against a distributed database of tuples that follow a relational schema. The query language of PIER is SQL. Query processing in PIER is supported through primary and secondary indexes. For each such index, PIER builds a table using a CAN DHT. Depending on the complexity of a relational schema and the number of indexes, maintaining a PIER index structure can be costly.

Frameworks for Unstructured Queries

Frameworks for unstructured queries typically support queries in form of keywords and match these onto file names or resource names. Peer-to-peer designs have been developed for such frameworks. Some of them are based on DHTs, some use a lattice to build indexes for query processing. In the following, we describe one example for each of these approaches.

pSearch is a framework that implements the Latent Semantic Indexing (LSI) model on CAN, which reduces the dimensionality of a traditional vector space model through principal component analysis [82]. In pSearch, both a query and an object, e.g., a file name, are represented as points in the CAN key space. A drawback of such an approach is that an LSI space typically still has a high dimensionality and leads to long query response times.

DPMS is a distributed, hierarchical indexing framework for processing keyword queries [83]. In DPMS, both queries and objects are represented as patterns, namely, $n$-grams (i.e., $n$ consecutive characters), and the query execution returns objects whose patterns are supersets of the query pattern. DPMS maintains a distributed index of object patterns on a network graph with lattice topology. The index
information of the nodes on the lowest of the lattice corresponds to the patterns
the object information available on those nodes. Index information is maintained in
form of a Bloom filter [84]. The index information of nodes not on the lowest level
is a Bloom filter that is computed by aggregating the bloom filters of lower-level
nodes. The DPMS design achieves redundancy through lower-level nodes sharing
their Bloom filter with several higher-level nodes. Query routing in DPMS works as
follows. When a node receives a query, it forwards the query to nodes on the next
lower level, if the query pattern matches the local Bloom filter. If there is no match,
the node forwards the query to a node on the next upper level. For a network of
nodes, query response time increases with $O(\log n)$, assuming the DPMS lattice,
the height of which increases with $O(\log \log n)$.

Frameworks for Spatial Queries

A spatial query retrieves spatial objects, such as points, lines, etc., from a universe of
objects by considering the spatial relationships, e.g., nearness, containment, etc., be-
tween spatial objects. Spatial objects are associated with locations or a subspace of
a multidimensional location space (often $\mathbb{R}^n$). Common examples of spatial queries
are the point query, which returns objects with a given location, the range query,
which returns all objects in a given range of a specific location, the nearest-neighbor
query, which returns $k$ objects nearest to a given location, and the centroid query,
which returns the objects at the centroid of a given set of locations [85, 12].

During the last decade, a number of peer-to-peer frameworks have been devel-
oped for processing spatial queries. Some of these frameworks use DHT-based net-
works for query routing [86, 87, 88]. Scrap and PRoBe are two well-known examples
[86, 87]. Both of these use the join and the leave operations of the DHT to construct
and maintain the distributed index. In the case of Scrap, the multi-dimensional lo-
cation space is mapped onto a single dimensional location space using a space-filling
curve. This single-dimensional location space defines an identifier space in the form
of a ring that is managed by a skip graph DHT [89]. The problem with this approach
is that Scrap does not guarantee the preservation of locality in higher-dimensional
location spaces. (The preservation of locality means that two locations that are
close to each other either share the same responsible peer, or their responsible peers
are neighbors.) For a range query, for instance, the object locations that satisfy the
query may not be found on a contiguous subgraph of peer nodes, and, therefore,
the query overhead may be large.

Second, in the case of PRoBe, the multi-dimensional location space is directly
mapped onto an identifier space of the same dimensionality, which is managed
by a CAN-based system [76]. While the PRoBe design preserves locality, query
processing can still result in high overhead and high latency due to the underlying
CAN design. This is particularly true when a peer invokes a point query for a
location that is far away from its hyper-rectangle (see above).

Apart from DHT-based indexes, other approaches have been developed to ef-
iciently process spatial queries, first for centralised and later for distributed sys-
tems. Among these approaches, the most widely used index structure is the R-tree
[90, 91, 92]. An R-tree is based on the geometric concept of the Minimum Bound-
2.5. GRAPH DATABASES

Ining Rectangle (MBR) [93, 94], which is a hyper-rectangle in a location space $L$. An MBR is defined by two locations, $(l_1, \ldots, l_n)$ and $(h_1, \ldots, h_n)$, whereby $l_i \leq h_i$. Given a set of locations, an MBR is the hyper-rectangle with the smallest diameter that includes all of these locations. Each node of an R-tree is associated with an MBR, whereby the MBR of a non-leaf node is the minimum bounding rectangle that includes the MBRs of all children. R-trees are used as index structures in spatial databases [95, 96, 97].

During the last decade, designs have been devised to use R-trees in peer-to-peer frameworks that underlie large-scale spatial information systems [96, 97, 98]. Two well-known examples are VBI-tree and DP-tree [96, 97]. These two works contain distributed algorithms to process spatial queries using R-tree based indexes. The indexes are built by distributing a given R-tree onto peer nodes. More details about VBI-tree and DP-tree can be found in [99].

Recently, a comprehensive peer-to-peer framework for processing spatial queries called MIDAS has been proposed [100]. MIDAS uses an index called k-d tree [101]. The main advantage of a k-d tree index is that the addition or removal of objects does not require a change in index. For a more in-depth discussion of the MIDAS design see [99].

Addressing the design challenges for network search using concepts from peer-to-peer frameworks

The fundamental problem of scalability in query processing in the design of a network search system can be addressed using concepts from peer-to-peer frameworks. First, for routing queries, the peer-to-peer frameworks discussed above provide mechanisms to implement a logical overlay on top of the physical network topology, where the nodes in the overlay form a subset of the nodes in the physical network. Second, these frameworks provide protocols for constructing/maintaining a distributed index structure on nodes in the overlay to support efficient query processing. Third, these frameworks provide distributed algorithms for processing search queries on a network graph. Many of these algorithms are scalable in terms of query response times and query processing overhead.

2.5 Graph Databases

Operational information in networked systems often has graph structure. An interesting aspect of network search, therefore, is modeling, searching for, and retrieving graph-structured information. From a database perspective, some of these issues have been studied in the context of graph databases.

A well-known graph data model adopted for many graph databases is the property graph model [102]. In a property graph, each vertex represents an information object, and each edge represents a relationship between two objects. Each vertex, as well as each edge, has a unique identifier and is associated with a set of properties expressed as attribute-value pairs. Further, a vertex is associated with a set of incoming/outgoing edges, whereas an edge is associated with a start-vertex and
CHAPTER 2. RELATED RESEARCH

an end-vertex. Examples of graph databases that are based on the property graph model are Neo4j [103], InfiniteGraph [104], and Titan [105].

A second well-known graph data model for graph databases is the Resource Description Framework (RDF) [53]. It is a standard for interchanging data over the web and it is part of the Linked Data framework [52]. Information in RDF is represented by an RDF graph, which is a set of subject-predicate-object triples. Subjects in such triples are uniquely identified by uniform resource identifiers [14]. Examples of graph databases based on the RDF graph model are AllegroGraph [106], Virtuoso [107], and Trinity [108].

SPARQL is the standard query language for retrieving data stored in the RDF format [109]. A typical SPARQL query is a set of triple patterns. A SPARQL triple pattern is similar to an RDF triple except that any element in the triple pattern can be a variable. Processing a SPARQL query is achieved through subgraph matching, whereby each pattern in the query is matched against a triple in all RDF graphs in the database. Examples of graph database platforms that implement SPARQL are AllegroGraph [106], Virtuoso [107], Trinity [108], and Titan [105].

Gremlin is a query language for retrieving data stored in the property graph format [110]. A Gremlin query can be seen as a tree of functions, whereby each function defines a traversal step in a property graph and returns vertices that either are the results of the query or starting points for the next function. In addition to functions for linear traversal and branching, Gremlin includes more complex traversal functions, such as recursive traversal, path traversal, etc. Two examples of graph database platforms that support Gremlin queries are Neo4j [103] and InfiniteGraph [104].

We give an overview of two distributed graph database platforms that support SPARQL queries over data stored in RDF format. The first platform, Virtuoso, stores a set of RDF graphs as a table with four columns that are labeled graphs, subjects, predicates, and objects [107]. Virtuoso treats identifiers of subjects as primary keys, which is the basis for partitioning a relation and maintaining the primary index for the table. To support accessing elements of RDF graphs, Virtuoso maintains four centralized indexes. It employs hashing to partition a table horizontally across cluster machines. To process a SPARQL query, Virtuoso translates the query into a plan with SQL operators. The plan is executed using SQL query processing techniques against a partitioned relational database, which performs well as long as the number of partitions is small.

The second platform, Trinity.RDF, provides a SPARQL interface on top of an in-memory distributed key-value store [108]. RDF triples are indexed as a set of key-value pairs distributed across cluster machines. In a key-value pair, the key represents the identifier of a subject and the value represents a list of predicate-object pairs drawn from the triples that correspond to the subject. A SPARQL query is represented as an ordered list of triple patterns. A variety of heuristics, such as estimations of unique subjects (or objects) associated with indexed terms, are used to determine the order. Processing a triple includes a traversal of the RDF graph through operators on the key-value pairs.

In order to design a system for searching graph structured data, the above works can be used as a starting point to identify an information model, a search query
language, and distributed query processing functions. The major challenge is to ensure query processing that scales to a very large system of 100,000 nodes, for which an efficient distributed index that can be maintained at low cost will be the key.

2.6 Bottom-up Designs of Networked Systems

When a bottom-up approach is used, the design process starts with the physical infrastructure, on top of which one or more layers are built, until the target functionality is realized on the highest layer. The most notable example of a bottom-up design is probably the Internet, which enables today global services for several billion users [111, 112]. The infrastructure layer of the Internet consists of access networks, local area networks, campus networks, the networks of Internet providers, etc. On top of this network infrastructure, efficient end-to-end communication services have been built. On a higher layer, Internet services have been developed that take advantage of underlying communication services. Generally, in all types of communication networks, including wireless networks, sensor networks, and vehicular networks, a bottom-up approach is applied to engineer application services.

In a similar way to networking, networked computing systems are often designed bottom-up. The lowest layer is generally an operating system, on top which a virtualization layer resides, which supports a large range of applications.

Bottom-up designs traditionally allow to optimize systems for performance objectives on the physical infrastructure, for example, maximizing resource utilization, minimizing execution times, or minimizing energy consumption. Note that peer-to-peer frameworks generally optimize performance for a logical overlay network. Since the goal of network search is efficiently retrieving infrastructure statistics, a bottom-up approach should be chosen.

2.7 Discussion

In this section, we surveyed several areas of research that influence our design of a network search system. During our work, we will review, evaluate, and possibly adapt some of the concepts outlined above. The design of a network search system must meet requirements that the surveyed information systems do not fulfill.

A key difference between network search and the above designs of information systems is that network search performs search against volatile and fast-changing operational data inside a networked system. This requires an architecture and a set of protocols for in-network, real-time query processing, which are not addressed by the above research works. As part of our approach, a bottom-up design will be needed. Such an approach will address the problem of building a distributed index structure from a network of local dynamic databases. In addition, a bottom-up approach will enable query routing on the physical network topology instead of on a logical topology, as the above works rely on.
Chapter 3

Summary of Original Work

The results of this research are documented in eight papers. The complete list is presented in Chapter 5. Four papers with the main results have been selected for this thesis.

Paper A: Management by Network Search

Misbah Uddin, Rolf Stadler, and Alexander Clemm.

While networked systems hold and generate vast amounts of configuration and operational data, this data is not accessible through a simple, uniform mechanism. Rather, it must be gathered using a range of different protocols and interfaces. Our vision is to make all this data available in a simple format through a real-time search process which runs within the network and aggregates the data into a form needed by applications—a concept we call *network search*. We believe that such an approach, though challenging, is technically feasible and will enable rapid development of new management applications and advanced network functions. The paper motivates andformulates the concept of network search, compares it to related concepts like web search, outlines a search architecture, describes the design space and research challenges, and reports on a testbed implementation with management applications built for exploratory purposes of the new paradigm.

*This paper has been published in proceedings of IEEE/IFIP Network Operations and Management Symposium (NOMS), Maui, Hawaii, USA, 16-20 April, 2012. It appears in this thesis as Chapter 6. My contribution to the work has been framework design and implementation. My supervisor Prof. Rolf Stadler and Dr. Alexander Clemm from Cisco Systems helped with fruitful discussions on all aspects of this work.*
CHAPTER 3. SUMMARY OF ORIGINAL WORK

Paper B: A Query Language for Network Search
Misbah Uddin, Rolf Stadler, and Alexander Clemm.

Network search makes operational data available in real-time to management applications. In contrast to traditional monitoring, neither the data location nor the data format needs to be known to the invoking process, which simplifies application development, but requires an efficient search plane inside the managed system. This paper presents a query language for network search and discusses how search queries can be executed in a networked system. The search space consists of named objects that are modelled as sets of attribute-value pairs. The data model is more general than the relational model, and the query language is more expressive than relational calculus. The paper shows that distributed query processing can be performed using an echo algorithm and that name resolution can be embedded in query processing. Finally, two use cases for network search are presented, one in networking and one in cloud computing, the latter backed up by a prototype implementation.

This paper has been published in the proceedings of IEEE/IFIP International Symposium on Integrated Network Management (IM), Ghent, Belgium, May 27-30, 2013. It appears in this thesis as Chapter 7. My contribution to the work has been concept development, algorithm design, and implementation. My supervisor Prof. Rolf Stadler helped with fruitful discussions on all aspects of this work. Dr. Alexander Clemm from Cisco Systems helped with formulating a use case, in addition to his valuable comments on all aspects of this work.

Paper C: Scalable Matching and Ranking for Network Search
Misbah Uddin, Rolf Stadler, and Alexander Clemm.
In Proc. 9th International Conference on Network and service management (CNSM), Zurich, Switzerland, October 14-18, 2013.

Network search makes operational data available in real-time to management applications. In contrast to traditional monitoring, neither the data location nor the data format needs to be known to the invoking process, which simplifies application development, but requires an efficient search plane inside the managed system. The search plane is realized as a network of search nodes that process search queries in a distributed fashion. This paper introduces matching and ranking for network search queries. We are proposing a semantic for matching and ranking, which is configurable to support different types of management applications—from exact matching for database-style queries to loose, approximate matching, which is appropriate for exploratory purposes. We describe an echo protocol for efficient distributed query processing that supports matching and ranking. Further, we present the design of a search node, which maintains a real-time database of operational information and
allows for parallel processing of search queries. A prototype implementation on a cloud testbed shows that the network search system, on a 9-node cluster with 24 core servers, executes 200 global search queries/sec with the 75th percentile latency below 100 milliseconds and with a CPU utilization below 5%. The performance measurements, together with our design, suggest that a system of 100’000 servers processing the same load would exhibit the same overhead per server and a query latency of below 1 sec.

This paper has been published in the Proceedings of 9th International Conference on Network and Service management (CNSM), Zurich, Switzerland, October 14-18, 2013. It appears in this thesis as Chapter 8. My contribution to the work has been concept development, algorithm design, search node design, implementation, and evaluation. My supervisor Prof. Rolf Stadler and Dr. Alexander Clemm from Cisco Systems helped with fruitful discussions on all aspects of this work.

Paper D: A Bottom-Up Design for Spatial Search in Large Networks and Clouds

Misbah Uddin, Rolf Stadler, and Alexander Clemm.
Submitted for publication.

Information in networked systems often has spatial semantics: routers, sensors, or virtual machines have coordinates in a geographical or virtual space, for instance. In this paper, we propose a design for a spatial search system that processes queries on spatial information, which is maintained in local databases inside a large networked system. In contrast to previous works in spatial databases and peer-to-peer designs, our design is bottom-up, which makes query routing network-aware and thus efficient, and which facilitates system bootstrapping and adaptation. Key to our design is a protocol that creates and maintains a distributed index of object locations based on information from local databases and the underlying network topology. The index builds upon the concept of the minimum bounding rectangle (MBR) to efficiently encode locations. We present a generic search protocol, which is based on an echo protocol and uses the index to prune the search space and perform query routing. By design, the response times of search queries increase with the diameter of the network, which is asymptotically optimal. Through simulation, we study the performance of the protocol in static, and dynamic network environments, for different network topologies, and for network sizes up to 100’000 nodes. In all simulation experiments, the overhead incurred by our protocol lies within thirty percent of a hypothetical optimal protocol. In addition, the protocol provides high accuracy under significant churn.

This paper has been submitted for publication. It appears in this thesis as Chapter 9. My contribution to the work has been concept development, algorithm design, and evaluation. My supervisor Prof. Rolf Stadler and Dr. Alexander Clemm from Cisco Systems helped with fruitful discussions on all aspects of this work.
Chapter 4

Open Problems for Future Research

Based on the work presented in this thesis, we have identified several research directions that merit further investigation.

1. *Global index for dynamic, operational data:* The search algorithms presented in Chapter 8 and 9, which are used to find objects with given operational states, have been shown to scale well. However, there is still room to improve their efficiency, since these search algorithms exhaustively touch all nodes in the network to identify such objects. The search algorithm that we developed for processing spatial queries (presented in Chapter 10) is efficient in terms of reducing the number of nodes that processes a query, since it uses a distributed index. Such an approach can be used to make the algorithms presented in Chapter 8 and 9 more efficient. Typically, in a networked system, the rate of change of volatile, operational data is much faster than that of spatial data. Therefore, using an indexing scheme for spatial query processing to index volatile, operational data in a networked system can incur overhead costs that may not be acceptable. A key challenge remains how our proposed indexing scheme can be extended to support query processing on fast changing data while keeping the overhead for index management acceptable.

2. *Graph search:* A large part of operational and configuration information of networked systems has graph structure, e.g., virtual network topologies, IP flows, communication links of distributed cloud applications, etc. An interesting problem is to develop a scalable management system that allows real-time management applications, such as network analytics and anomaly detection applications, to search for graph-structured operational information. The key challenge is to devise an efficient and scalable search process on data that is volatile and distributed across the network infrastructure. Similar to the approach taken to devise an efficient spatial query processor in our work (see Chapter 10), one can investigate to which extent a global search index can be applied to devise an efficient graph search system. We presented early results in designing a graph search system in [113]. One can build upon our work.
3. **Persistent search queries:** For management applications, persistent search queries can be useful, since they allow applications to react dynamically to changes in network states. Stream processing engines can provide the basis for enabling persistent search queries. One avenue of research is to study how the technology of stream processing can be used to develop a query processor for persistent search queries.

4. **Search on historical data:** For many management applications, such as the ones that rely on network analytics, access to historical data is important. It is worth investigating how our current model and algorithms can be extended to support search on historical information.

5. **Multi-domain search:** A network search system can span multiple domains. In that regard, how to support global search services across multiple domains with local administrations remains to be answered.

6. **Security and Privacy:** Security and privacy is a concern in any service deployed on networked systems. Right now, our design of network search is agnostic to security and privacy concerns. Studies are required to understand secure search services and queries with different access privileges.
Chapter 5

List of Publications in the Context of this Thesis


Chapter 6

Management by Network Search

Misbah Uddin†, Rolf Stadler†, Alexander Clemm‡
† KTH, Royal Institute of Technology, Sweden. Email: {ahmmud,stadler}@kth.se
‡ Cisco System Inc., San Jose, USA. Email: {alex}@cisco.com

Abstract

While networked systems hold and generate vast amounts of configuration and operational data, this data is not accessible through a simple, uniform mechanism. Rather, it must be gathered using a range of different protocols and interfaces. Our vision is to make all this data available in a simple format through a real-time search process which runs within the network and aggregates the data into a form needed by applications—a concept we call network search. We believe that such an approach, though challenging, is technically feasible and will enable rapid development of new management applications and advanced network functions. The paper motivates and formulates the concept of network search, compares it to related concepts like web search, outlines a search architecture, describes the design space and research challenges, and reports on a testbed implementation with management applications built for exploratory purposes of this new paradigm.

6.1 Introduction

Any networked system holds and generates vast amounts of configuration and operational data in configuration files, device counters and data collection points. This data is partitioned in the sense that it is kept in various formats and needs to be accessed through different protocols and interfaces, including SNMP, NetFlow, CLI, etc. By making all this data available to a general search process in real-time across the network—a concept we call network search—novel management applications and advanced network functions can be envisioned.

Network search can be seen as a functionality that extends or complements monitoring, whereby state information and monitoring variables are not explicitly
addressed by network location but characterized as content. While we envision that the concept includes searching static and dynamic data, current as well as historical information, we focus in this paper primarily on dynamic data, as we see the most innovative aspects in this area. Another way of understanding network search is as “Googling the network in real-time.” For instance, the search query “find information about the flow <src, dest, app>” may return a list of systems that carry the flow in their caches, the interfaces the flow uses, the list of firewalls the flow traverses, the list of systems that impose traffic policies on this flow, etc.

We anticipate that the capability of network search will accelerate the creation of novel network management functions. It will spur, for instance, the development of new tools that will assist human administrators in network supervision, maintenance and fault diagnosis. The tools will allow search queries, such as:

- **Search for information about the system with IP address a.b.c.d** which may return links to systems that list it as BGP neighbor, to firewalls or NATs that include its address, or to event logs where its address appears.

- **Search for excessive resource utilization**, which may return links to systems and components whose CPU, memory or communication links are highly utilized.

- **Search for information about a session with given endpoints**, which may return the paths of flows associated with the session and information about the supporting virtual and physical infrastructure

We believe that a network search functionality will allow the rapid development of new classes of network control functions and applications, for instance:

- **Tracking virtual networks and virtualized resources**, whereby virtual systems and resources are dynamically discovered and their states, physical and logical dependencies, and configurations tracked.

- **Service level diagnostics**, whereby critical session flows that are subject to service level agreements are discovered and analyzed in real time for potential detriments to performance.

- **Peer ranking**, whereby hosts are ranked for peer selection in the context of peer-to-peer applications peering, based on the hosts’ locality, distance, lifetimes, etc., which is discovered and collected through the search process.

While it is certainly feasible to engineer the above functions with available technology, they require a considerable effort in devising the individual subsystems that identify, localize, and collect the data in a networked system in real-time. Our point is that a general network search functionality would significantly accelerate the development time for such functions. Today, their development struggles not only to keep up with ever-increasing heterogeneity, but also to deal with the increasing volume of data that network elements are able to generate.
In the past monitoring generally involved occasional polling of key MIB objects, as well as listening to traps and syslog messages. Today, however, often large amounts of Netflow data are exported, with numerous per-flow statistics, and we anticipate that services will soon take periodic snapshots in rapid succession of most operational state. This trend makes real-time exploration and analysis of data in a networked system increasingly costly and requires significant infrastructure outside the network.

Network search will require more resources than monitoring, as the search process involves discovery, not only data retrieval from known location. However, search can be coupled with aggregation of partial results, which allows some degree of processing of data as part of the search process. To achieve scalability and fast response times to queries, we believe that network search must be performed inside the network infrastructure, supported through a decentralized, self-organizing search architecture.

Recent advances in research, as well as in emerging technologies, when leveraged properly, make network search feasible for today’s network environments. The development of scalable search functions can draw upon results in resource discovery, in-network aggregation, distributed real-time monitoring, and scalable storage—often using tree-based or gossip-based distributed algorithms—which have been produced over the last decade by the networking, management and distributed systems communities. Also, query processors have been developed for very large databases, as exemplified through Google’s Dremel system[114]. The instrumentation that network search requires has not been supported by off-the-shelf technology until very recently, since networking devices were considered closed systems, and management functions were provided off-box.

Initially driven by the need to automate management workflows, e.g., through supporting scripts in network elements that are triggered by network events, devices are increasingly offering APIs that can be used to significantly extend their functionality. Also, additional processing capacity is becoming available within network elements, for instance in form of service blades, such as Cisco’s Network Analysis Module (NAM), originally used for passive measurements, deep packet inspection and real-time traffic analysis, but programmable for other uses. Furthermore, with the proliferation of multi-core CPUs on network blades, some cores are available for other purposes than traffic forwarding and signaling.

The contributions of the paper are as follows. We motivate and formulate the concept of network search and discuss related concepts. We outline an architectural framework for a distributed search system and describe its design space and associated research challenges. We report on a testbed implementation set up for exploration of this paradigm, where a simple search system with a search interface for an administrator has been built, together with an application for detecting traffic anomalies, which relies on network search functions.

Section 6.2 of this paper discusses concepts and work related to network search. Section 6.3 provides a proposed architectural framework for network search, and Section 6.4 presents a testbed and a prototype we built for exploratory research. Section 6.5 discusses the design space and challenges of realizing network search functions, while Section 6.6 presents the conclusions.
6.2 Related Concepts

On a conceptual level, network search has similarities with the general field of Information Retrieval, which deals with searching for documents that match certain criteria in a large document space [16]. Each document is represented by an index, and a search query is matched against all indexes of the document space using a query processing system. The result of a query is a ranked list of indexes, whereby a higher ranked index indicates a closer match of the index with the query.

Web search can be seen as an application of information retrieval technology to the web. As exemplified by Google’s search engine, web search is performed by matching keywords against the universe of web pages. Search results are presented as ranked lists of links to web pages, which can be accessed through a web browser. Matching is performed on a distributed inverted index of the web pages, using a dedicated search infrastructure outside the web. The index database is populated through so-called crawlers, which continuously navigate the web following the hyperlinks, i.e., the links between pages [27]. The ranking of the pages is computed through keyword matching and through the analysis of their hyperlink structure, for which several techniques have been developed, including HITS [115], PageRank [27]. More recent approaches consider the search history of social groups for computing the ranking [41]. With the advent of social tagging systems, the concept of web search has been extended. In addition to web pages, the search space contains now people, movies, blogs, etc., and, in addition to hyperlinks, the ranking considers relationships, recommendations, sharing of links, etc. [39].

Recently, the search process has been adapted to enable live search on news, blogs, forums, and the like. Instead of using crawlers, the search database is updated through RSS feeds [35, 116]. The ranking of results in live search is computed differently from the ranking in traditional web search and uses reliability of sources, trends in keywords popularity, locality of information, freshness of data items, and similarity of objects. A concept for live search, which researchers from Google call “query-free-news search” and which primarily computes the rankings according to object similarity, is presented in [117]. An example of a search system that ranks blogs using several of the above mentioned criteria is BlogScope [116].

Over the last years, the research area in the Internet of Things, which investigates the internetworking of physical entities, which generally contain sensors and actuators, gave rise to an effort called Web of Things, which includes adapting of web search to search within the space of Internet of Things. Entities in the Web of Things are realized as web pages, which means they can be identified through URLs and accessed through web protocols. Further, these entities span a wide range of real-world objects, from rooms in buildings to mobile phones. These entities contain sensors and thus have associated quantities, from occupancy levels, over temperature, video captures to locations. A survey of concepts in the Web of Things and their implementations can be found in [45].

While it is possible to use traditional web search systems to search the Web of Things, additional matching schemes have been developed. They include comparing measured quantities against specific values, allowing to search, for instance, for an empty room in a building or the last known location of a cell phone. In addition,
specialized ranking schemes have been devised. The search system is generally realized as a hierarchy of processing nodes, which perform the search and collect the results. Various implementations in the contexts Web of Things are presented in [47, 50, 48, 49, 46].

An active, established field that relates to network search is the area of query processing in very large databases. A key topic of this research is assuring the scalability of query response times, which can be achieved through various forms of weak consistency or through optimizing the system for certain types of queries. The Dremel system is an example of the latter [114]. It enables fast response times for queries on tables with large numbers of columns, whereby only a small number of column attributes appear in the query. Queries are executed in a distributed fashion on servers that hold data, which is uniformly structured and potentially replicated. In [114], a prototype is reported on, which executes within 20 seconds a query on a single attribute in a database of 85 billion records that are horizontally partitioned over 3000 nodes.

The process of mining event logs from a network is related to network search. Event logs are created by network elements and collected in data collection points. Mining tools process either a database of event logs or a real-time stream of events. The processing is generally performed in a centralised way, whereby one mining process acts on one data store. Data mining is often used on event logs for anomaly detection. In [118, 119, 120], for example, frequent itemset mining is applied (a) to netflow logs in order to identify network hotspots and (b) to network IDS logs in order to classify network alerts in real time.

Having reviewed concepts underlying, and technologies developed for, Information Retrieval, Web Search, Web of Things, very large databases, and data mining, we ask the question: to which extent we can these be applied for the purpose of engineering a network search system?

Information Retrieval offers an approach where a document is represented as an object of simple structure and characterized by an index. The search language is limited, and a search query is expressed using index terms. The search semantic is captured in the matching and ranking functions. Information Retrieval is applicable to domains with a large number of objects. It offers a simple query language, and queries can be efficiently executed over large data sets. What makes Information Retrieval potentially attractive for network search is the large number of network objects that could be covered. The downside is that those objects must be of simple structure.

Web Search and its extensions discussed above offer an approach to information retrieval whereby the object space is distributed across the Internet. Sophisticated versions of matching and ranking functions have been developed to reduce the result set of search queries, which may be helpful in developing network search concepts. In addition, protocols and tools have been engineered that are of potential interest to network search developers. In the context of Web of Things, strategies have been devised to allow for distributed search, by distinguishing between index nodes and data collection points, for instance, which may be helpful in designing a scalable network search system.

The field of very large databases, as exemplified above through concepts realized
in the Dremel system, has made advances in scalable query processing for certain classes of queries. Such queries, which are based on relational algebra, can be much more expressive than Information-Retrieval-type queries, at the expense of a higher computational complexity for query processing.

Log mining can provide a higher level of abstraction or more complex processing of data than database processing and certainly Information Retrieval can. The difficulties of directly using log mining techniques for network search are that these techniques generally rely on a centralized data store and are not designed for real-time use.

6.3 A Framework for Network Search

Modeling and querying network information

For the purpose of modeling and querying network information, we advocate using concepts that underlie the above reviewed fields of Information Retrieval, Web Search and Web of Things. Our choice is motivated by the objective of having a network search capability suitable for real-time search in a vast object space that is distributed over a large networked system. In addition, we want to make data from legacy systems available for search in a straightforward, efficient way, and we want the search system to adapt to a changing network and easily cover new network elements. The price we pay for our choice of model includes losing structural information of objects and accepting a less expressive query language than, e.g., a database approach would provide. Compared to a query expressed in an SQL-type database language, a similar query in a network search language will often produce only an approximate result, as it is less expressive.

Following our approach, designing a model for network search includes devising a simple object model, a concept for an object index, and a scheme for object naming. In addition, a query language must be developed, together with matching and ranking functions, which define the semantics of the query process.

In a simple prototype, further described in Section 6.4, we implemented a model in which a network object, as well as its index, is expressed as a bag of attribute-value pairs. A query is expressed as a list of tokens, whereby each token contains an attribute-value pair, a comparison operator, and an optional aggregation function. A query returns those objects whose attribute-value pairs match all tokens of the query. The returned result is in aggregated form, if the query contains an aggregation function. The simple, restrictive search language includes neither object names nor a ranking function, although these may be included in the future.

An architecture for network search

Our architecture for a network search system is depicted in Figure 6.1. Its key element is the search plane, which conceptualizes the network search functionality. This plane contains a network of search nodes, which have processing and storage capacities. A search node can communicate with a set of neighbors, which are identified through links of the network graph. The design of this plane supports
6.3. A FRAMEWORK FOR NETWORK SEARCH

Figure 6.1: Proposed architecture for a network search system

searching in a distributed and parallel fashion. A search node can be realized in various ways: it can be part of the management infrastructure outside the managed system, it can be run as a standalone network appliance, or it can be integrated into a network element using a variety of technologies. Our current prototype implements the second option, while we envision the third option for future realizations of network search systems.

The bottom plane in Figure 6.1 represents the physical network that is subject to search. Each network element is associated with a search node, which maintains (or has access to) configuration and operational data from that network element. The top of Figure 6.1 shows the management plane, which includes the systems and servers running processes for network supervision and management.

There are two important interfaces in this architecture. The first is the query interface, which allows a process in the management plane to execute a network search operation. We envision that every search node is an access point for such a query. The second interface defines the interaction between a search node and a network element, which can be realized through polling or can be push based. This interface is technology-dependent and possibly proprietary.

Algorithms and functions in the search plane

Each search node runs a process that communicates with the associated network element(s) from which it retrieves data in a networked system. A database function dynamically maps that data into the information model for network search and updates the local search database.

Search functions, invoked from the management plane through query invocation,
are executed as distributed algorithms on the graph of search nodes. During the execution of a query on a search node, the local search database is accessed, the matching of the local query against stored indexes is performed, and the local search result is possibly aggregated with results from other nodes.

We envision that network-wide abstractions of data, such as traffic matrices and other global objects, are dynamically constructed and maintained by processes in the search plane which aggregate data from local search databases across time and space and which store the results in the same databases, thereby making network-wide abstractions available to search queries from the management plane.

To develop distributed search functions, results from very large database research are potentially helpful. Furthermore, research efforts in the fields of Web Search and Web of Things have produced many approaches to query matching, result ranking and result aggregation, which are useful developing scalable search functions.

6.4 A Platform for Network Search

The testbed infrastructure

Our testbed for experimentation, which is part of a larger laboratory infrastructure, includes 16 Cisco 2600 Series routers and 33 Hosts interconnected via four 100 Mbps FastEthernet switches. The routers run the OSPF protocol for routing. All hosts are rack-mounted Pentium-4 computers with a 2.8 GHz CPU and 1 GB RAM, running Ubuntu 10.04. 16 of the hosts are configured as traffic generators, which produce load using pktgen [121]. In addition, they use the fping [122] and hping [123] packages to inject anomalous traffic into the network.

The network search prototype

The routers in the testbed form the network plane on which search is performed (see Figure 6.1). With each router, a host is associated running a search node. There is one search node per router. The management plane currently includes a single management station on a dedicated host. Search nodes communicate with one another and with the management station through the routers in the network plane.

The search nodes dynamically populate their local databases with configuration data from the routers, such as interface descriptions, IOS version, and routing table (which change at slow time scales) as well as with operational data, such as network statistics from MIB variables and IP flow statistics from NetFlow caches (which change at fast time scales). The data is collected by polling SNMP MIB objects or by issuing CLI commands, at a rate reflecting the frequency of change of particular data items.

A database module on the search node converts the collected data to the object model of the search database described in Section 6.3. For example, the text line representing an IP flow, “192.168.1.1 : 3171, 10.10.10.10 : 21, 6, 3, 284” is mapped onto a bag of attribute-value pairs be expressed as “{(srcIP, 192.168.1.1),
The search database is kept in the memory of the search node to allow for fast access. It is organized as an inverted index and maintained using the indexer libraries of the Apache Lucene package [124].

A query module on the search node matches queries invoked on the management station with the content of the local database and returns the results. In our current implementation the results are not ranked. The query module uses Lucene’s query libraries.

The network search function is realized in a simplistic way. Its functionality is centralized, and it executes on the management station. It is made up of two modules—the communicator and the aggregator. When the communicator receives a query from a management application, it forwards the query to all search nodes. The responses are then processed by the aggregator module, and the result is handed to the application.

The modules of the search node are written in Java and shell script, and those in the management station in Python.

**Application 1: A simple network query language**

We have implemented an interpreter for a simple network query language, which allows a management application to search for information in the network. The query language is defined on the data model introduced in Section 6.3 and characterizes the interface between the management plane and the search plane. The version of the language given here is sufficient to explain its basic concepts and for exploratory use on our testbed. For operational use, however, it is too simplistic and will need to be extended, to include object names and ranking functions, for instance.

We describe the query language through examples. One simple query is `search a`, which returns all attribute-value pairs in the search space whose attribute name matches `a`. The query `search a, b>100` returns all tuples of attribute-value pairs from bags that contain attribute `a`, and attribute `b` with a value larger than 100.

Per default, a query is executed as a one-time operation and returns a snapshot of the network state. A query can also be invoked in continuous mode, whereby the result is periodically recomputed. Further, a query can be invoked with an option to return the number of tuples (by adding a `-c` flag), or the number of unique tuples (by adding a `-u` flag), instead of the items themselves.

The interpreter for the query language is written in Python [125]. In our current setup, the interpreter runs on the management station, which processes a query by sending it to all search nodes, aggregating their responses, and returning the result to the invoker.

Here are some examples of queries we can invoke on our testbed.

**Query 1: Search for the router with the highest load.**

The query `search router, load` is invoked in a Python script. The query returns tuples of attribute-value pairs that contain the IP addresses of all routers in the network and their respective ifInOctet values. Then, the script identifies the router with the highest value.

**Query 2: Search for flows that pass through two given routers.**
The query `search router=a, srcIP, dstIP, srcPort, dstPort, Protocol` and the corresponding query for `router=b` are invoked in a Python script. Each query returns a set of six-tuples, one for each IP flow traversing the routers `a` or `b`, respectively. The script then computes the result by intersecting the two sets.

**Query 3: Search for potential spammers.**

Two queries `search -c srcip, srcport=25`, and `search -c dstip, dstport=25` are invoked in a Python script. They return the number of outgoing and incoming SMTP flows of known hosts. The script then computes the ratio of the two numbers for each host and raises an alarm for those hosts whose ratio is above a certain threshold. The identified hosts need to be further investigated.

The above examples illustrate that, in order to search for data in a networked system using our language, neither the detailed structure of the information nor the location of the data items must be known. What we must provide as input to a query are solely attribute name(s) and value(s) that characterize the information we wish to retrieve.

**Application 2: Network-wide anomaly detection**

In this subsection, we present an application built on top of the simple search interface described in Section 6.4. The application is designed to detect network-wide anomalies, and we use it in experiments to find anomalies related to simulated network attacks on the testbed.

The application is written in Python and makes use of several open-source software packages. It uses operational data associated with network flows and link utilization.

The anomaly-detection application periodically executes a series of steps as follows. Using the network search functionality, it queries the search plane for flow and utilization data. A Principal Component Analysis (PCA) component reduces the dimensionality of this data to two dimensions. Then, a clustering component groups the data points into recognizable clusters. After that, unusual clusters, i.e., clusters that are far from the center of gravity of all data points, are identified. Then, a data mining technique, implementing a frequent-itemset-mining scheme, identifies flow patterns that are associated with the unusual clusters. Finally, these flow patterns are matched against possible attack signatures.

For performing PCA and clustering, the application makes use of FactoMineR, an R-based open-source tool for multivariate data analysis ([126, 127]). For performing frequent itemset mining, it relies on the LogHound package [128].

We illustrate the operation of the anomaly-detection application through an experiment on our testbed. We inject TCP traffic into the network, which we consider a “normal” traffic pattern for the purpose of the experiment, using the traffic generators equipped with pktgen. In addition, we periodically inject a sequence of three traffic patterns, each of which simulating a network attack—a ping sweep (for 5 sec), a port scan (for 3 sec) and a synflood (for 3 sec) ([129, 130]), which are some 30 sec to a minute apart from one another. The attacks are created using two network analyzing tools, fping [122] and hping [123].
During the experiment, a search query is executed every second, its result set is aggregated into a vector of 17 components, and the analysis is performed over a window size of the latest 20 such sets. Figure 6.2 gives the output of the FactoMineR module at four distinctive times during the experiment. It shows the projection of network states in which clusters 1 and 3 are marked as normal, and 2 and 4 as unusual. Figure 6.2(a) refers to the network under normal operation, in which clusters 1 and 3 are marked as normal, and 2 and 4 as unusual. Figure 6.2(b) refers to the network during a port scan, in which cluster 2 is marked as normal, and 1 and 3 as unusual. Figure 6.2(c) refers to the network during a Denial of Service (DoS) attack, in which clusters 2 and 3 are marked as normal, and 1 and 4 as unusual. Figure 6.2(d) refers to the network during a ping sweep, in which clusters 1 and 3 are marked as normal, and 2 and 4 as unusual.

After identifying the unusual clusters, the application invokes the loghound module, which performs frequent itemset mining on the flow data associated with the data points in the unusual clusters. The module returns flow patterns of high frequency. In the data associated with Figure 6.2(a), it identifies several patterns, none of which matches any known attack signature. In the data associated with Figure 6.2(b), the module detects several patterns, for example srcIP:192.168.5.150 dstIP:192.168.2.150 * proto:06 pkt:1 freq:9.3%. This pattern is consistent with the signature of a port scan, because it represents a large number of flows, each consisting of a single packet, sent to different ports of the same node. Similarly, the module detects patterns from the data associated with the unusual clusters in Figure 6.2(c) and (d), which are consistent with the signatures of a DoS attack and a ping sweep, respectively.

During the course of this experiment, our application correctly detected anomalies related to the three simulated attacks.

The application performs an analysis of the network state every second and produces an alarm, if an anomaly is detected, within some 250 milliseconds from the start of an analysis cycle.

### 6.5 Design Space and Challenges

The task of engineering the search plane introduced in Section 6.3 and building novel applications, such as those described in Section 6.4, opens up many interesting problems. Due to lack of space, we limit ourselves to the design space and research challenges associated with devising efficient search algorithms. We leave out other issues, including searching in a multi-domain environment, privacy aspects of search data, securing the search infrastructure, handling search nodes with different capacities, etc., which will be discussed elsewhere.

The design goals for a search algorithm are short execution time, low overhead in consuming search plane resources, and scalability, which means sub-linear growth of these two metrics with increasing system size. In addition, a search algorithm should dynamically adapt to changes in the network configuration and provide results with high precision. Obviously, these metrics cannot all be jointly optimized,
Figure 6.2: Four snapshots of the network state at different times of the experiment. The dimensionality of the state-vectors from network search are reduced to two dimensions using Principal Component Analysis. The state-vectors are then clustered into a small set of recognizable groups, using a hierarchical clustering method. Clusters far from the center of gravity are identified as outliers, which need to be further investigated. (a) refers to the system under normal operation, (b) to a DoS attack, (c) to a ping sweep, and (d) to a port scan.
and, therefore, the tradeoffs need to be studied, and engineering solutions need to be developed to make them controllable.

A naive approach to search is flooding the search plane with local queries, possibly using a gossip algorithm. To avoid the initiator of the search query from being overloaded with answers from individual search nodes, a wave algorithm, such as echo, will likely perform better, as it allows aggregating the partial results using a spanning tree [131]. To further reduce the search overhead, exploiting domain-specific knowledge to guide the search process and applying heuristics to restrict the search space may prove effective. For example, knowing that IP addresses are subject to firewall rules, that they can designate source and destination addresses of flows, that they are associated with systems which have names and aliases, that they are assigned by DHCP servers out of address pools, etc., may help speeding up the processing of certain queries. When a query contains an IP address as a search term, the search algorithm may check Access Control Lists and flow entries and follow entries to the next hop a flow gets routed to, or the algorithm may check a system's DNS server for corresponding domain names and the DHCP server for information about the IP’s lease. Similarly, for a query with a parameter IP address, knowledge about the network topology and routing, both of which can be dynamically acquired, can be used to propagate the query and bound its search space.

When engineering the search plane for efficient operation, the dynamics or life times of data in a networked system must be considered. Information related to physical system configuration or installed software licenses is fairly long-lived, which allows caching. Other information, on short-lived flows for instance, does not, as is changing too fast. (Certain local statistics may even be computed on demand, as continuous updates may be too expensive.) A related problem is the placement of index data, against which queries can be executed. The fact that much data is transient suggests that indexes should be kept on or close to the search nodes. Centralizing and replicating index data to a certain degree will shorten the execution of queries, while, at the same time, will increase the resources in the search plane needed for updating the index. A possible approach that allows to control this tradeoff involves maintaining distributed index trees in the search plane, whereby individual indexes are pushed towards the root, depending on their particular lifetime.

6.6 Discussion

In this paper, we motivated and introduced the paradigm of management by network search. The paradigm addresses the problem of diversity in monitoring interfaces, by introducing a search mechanism that allows uniform access to data in a networked system in a simple format and in a way that is oblivious to the data location. The traditional “precise” monitoring interfaces are replaced by a single “less precise” query interface. The implication of this type interface on the type of management tasks that are particularly suited for the paradigm needs further investigation.

Leveraging technology trends that allow more than ever customized in-network
processing of management information, we advocate that data is accessed and aggregated inside the network, which can reduce the infrastructure needed today for processing monitoring data outside the managed system.

Having access to a search query interface, as described in this paper, can accelerate the speed of developing management applications, specifically for those applications that require data from various sources at potentially unknown or changing location, possibly in aggregated form. We expect the emergence of novel solutions to applications with these requirements.

From the experimental work described in this paper, we draw the following conclusions. Comparing our concept of a network-search language to an SQL-based query language for networked systems [132], we note that queries in our language are stated in a simpler and “freer” form, without knowledge of the global schema. On the negative side, our approach cannot directly use the query processing framework that has been developed for SQL-based languages, which allows to efficiently process many classes of queries in a large-scale networked system [132]. Second, using public-domain software packages, we were able to develop a complex application for anomaly detection within a short period of time (approximately two weeks). A significant factor that made this possible was the availability of a network search system (although a very primitive one), which gave us uniform access to operational data.
Chapter 7

A Query Language for Network Search

Misbah Uddin†, Rolf Stadler†, Alexander Clemm‡
† KTH, Royal Institute of Technology, Sweden. Email: {ahmmud,stadler}@kth.se
‡ Cisco System Inc., San Jose, USA. Email: {alex}@cisco.com

Abstract

Network search makes operational data available in real-time to management applications. In contrast to traditional monitoring, neither the data location nor the data format needs to be known to the invoking process, which simplifies application development, but requires an efficient search plane inside the managed system. This paper presents a query language for network search and discusses how search queries can be executed in a networked system. The search space consists of named objects that are modeled as sets of attribute-value pairs. The data model is more general than the relational model, and the query language is more expressive than relational calculus. The paper shows that distributed query processing can be performed using an echo algorithm and that name resolution can be embedded in query processing. Finally, two use cases for network search are presented, one in networking and one in cloud computing, the latter backed up by a prototype implementation.

7.1 Introduction

Network search—or search in networked systems—can be understood in three ways. First, as a generalization of monitoring whereby the monitoring data is retrieved by characterizing its content in simple terms, without giving location or detailed structure of the data. Second, it can be understood as “googling the network” for operational data, in analogy to “googling the web” for content. Third, network search can be seen as a capability that views the network as a giant database of operational and configuration data, which can be queried through a database-like interface.

In this paper, we follow the database interpretation and develop further the concept of network search, which we motivated and introduced in [1]. Specifically,
we view data in a networked system as objects with a simple structure: an object has a (globally unique) name, a type, and a variable number of additional attribute-value pairs. Objects are linked through joint attribute-value pairs through which associations can be expressed and discovered. We introduce a language to express search queries. It turns out that processing search queries can be performed through similar techniques as query processing in distributed (relational) database systems; we propose a tree protocol that dynamically creates spanning trees inside the networked system and incrementally aggregates the partial search results, which are sent from the leaves of the tree towards the root. Object name resolution is embedded in a “natural way” in query processing.

While some database concepts help in engineering a network search system, there are clear differences between querying a traditional distributed database and performing search in a networked system. First, a search result may not be an exact match, but only ‘close enough’ to be returned by a query. Second, similar to web search, search results are ranked, according to how closely a specific object in the result matches the query, the search history, etc.

We believe that an important application area for network search will be capturing and tracing dynamic service behavior across time and space. For instance, network search can be used to identify and trace media streams associated with a videoconference across the nodes of a network, or to find the set of virtual machines associated with a particular application in a server cluster. The paper will provide more details about these use cases. Such functionality can, of course, be “hardcoded” beforehand in specialized protocols; network search, however, allows us to dynamically introduce such capabilities into a networked system.

The paper is organized as follows. Section 7.2 discusses related work. The overall framework for a network search system is shown in Section 7.3. Section 7.4 contains the proposed data model. Section 7.5 describes the query language, and Section 7.6 explains how search queries can be processed in a distributed fashion in a networked system. Section 7.7 discusses two use cases in some detail, and Section 7.8 reviews the paper’s contribution and presents future work.

7.2 Related Work

There are two main research areas that relate to the topics addressed in this paper: web search and its evolution and query processing in large networked systems. For further research areas related to network search, see related works section in [1].

Web search, as exemplified by Google’s search engine, is performed by matching keywords against the universe of web pages [16]. Search results are presented as ranked lists of links to web pages, which can be accessed through a web browser. Matching is performed on a distributed inverted index of the web pages, using a dedicated search infrastructure outside the web. The index database is populated through so-called crawlers, which continuously navigate the web following hyperlinks, i.e., the links between web pages [27]. The rank of a matched page is determined by several metrics, including keyword-relevance score [37], which measures the relevance of the keywords for the page content, the authority score, which mea-
sures the level of connectedness of the page in the hyperlink graph [115] [27], and page statistics [29].

As in web search, the search space in network search is very large. Objects in network search tend to be more dynamic and can have short life time than web pages, which requires a different architecture for network search, that allows for query processing close to the data (see Section 7.3). The concept of ranking is important in both areas. It has been very well developed in web search, while in our work on network search, it is part of our future plans.

Web search has evolved to include so-called live search on news, blogs, microblogs, etc. Live content expires at a faster rate than regular web content [36, 133]. To maintain the index structure for search, new techniques have been developed, including partial indexing [34], RSS feeds [35, 36], adaptive periodic crawling [37]. In addition to the metrics mentioned above, the freshness of the data is a determining factor in the ranking of search results [36, 38, 34]. Note though that the overall architecture of web search has not changed with the introduction of live search. In particular, matching is performed in a dedicated infrastructure outside the web.

A second innovation in web search has been annotating web pages with names of real-world or abstract entities, which has been initiated by Wikipedia [134]. The concept of annotated web has triggered research activities into searching for relationships between entities in web pages, for example, the fact that person x collaborates with person y. Two approaches are being pursued. First, an approach whereby such relationships are explicitly defined and then queried [135]; second, an approach where relationships are discovered through language analysis [136].

We believe that the concept of discovering relationships between objects is also important to network search. The language introduced in this paper allows to discover links between objects which have not been explicitly declared.

A recent focus of research has been search in web-based social networks. Social networks have concepts like friends and groups and allow actions such as sharing, liking, recommending etc., also known as social tagging [39]. Search in this context are guided by two principles. It is user-centric [40] [41] and it uses social graphs as a means to guide the search process in a vast search space [42, 43].

The idea that domain-specific knowledge can reduce the search space is applicable to network search. For instance, when searching for information that involves a flow, the search process may progress along the path of the flow, as indicated in the next-hop field in the flow record. By doing so, we restrict the process from exhaustively searching the space to searching along a path.

The concepts proposed in this paper relate to the field of query processing in large database systems, which has been recently driven by technology companies that maintain large-scale ICT infrastructures, such as Google and Yahoo. These systems are characterized by a large number of servers that form the nodes of a distributed database, which stores logs from operational data, web content, etc. The relational data model has proved to be too rigid for these databases, and relaxed models, also called non-SQL databases, have been developed. Examples include Google’s Dremel database system [114], together with the BigQuery query language [137]. Another example is Yahoo’s Pig database system [138], together with the PigLatin query language [139]. Academic works include the ASTERIX.
As advocated by the above described research, we find that the relational model is too strict for our purpose and that the expressiveness of the relational algebra should be retained in the query language. The difference between a database system as described above and a network search system is their respective objective: a database system retrieves known data, while a network search system discovers potentially unknown data and relationships. Furthermore, in contrast to querying databases, ranking is important to network search, as well as imprecise results are useful, if they are obtained at low cost.

To realize continuous network search queries, the active research field of distributed stream processing is relevant. Queries in stream processing systems are executed on a network of processors, whereby sub-queries run in the processors, and data streams are pushed through the network. Examples of academic research in this field includes TAG [141], and Borealis [142], whereas IBM’s System S [143] and Yahoo/Apache’s S4 [144] are well known industrial efforts.

### 7.3 An Architecture for Network Search

Figure 7.1 shows an architecture for a network search system, which we introduced in [1]. Its key element is the search plane, which conceptualizes the network search functionality. This plane contains a network of search nodes, which have processing and storage capacities. A search node can communicate with a set of neighbors, which are identified through links of the network graph. The design of this plane supports searching in a distributed and parallel fashion. A search node can be
realized in various ways: it can be part of the management infrastructure outside the managed system, it can be run as a standalone network appliance, or it can be integrated into a network element using a variety of technologies. Our current prototype implements the third option.

The bottom plane in Figure 7.1 represents the physical network that is subject to search. Each network element is associated with a search node, which maintains (or has access to) configuration and operational data from that network element. This data is modeled as a set of objects, whose structure is described in Section 7.4. Note that the figure shows the simplest form of association between a network element and a search node; it is possible that a search node maintains data from several physical devices, or, alternatively, a device updates data on several search nodes. The top of Figure 7.1 shows the management plane, which includes the systems and servers running processes for network supervision and management.

There are two important interfaces in this architecture. The first is the search interface, which supports the query language discussed in Section 7.5. We envision that every search node is an access point for search queries. The second interface defines the interaction between a search node and a network element, which can be realized through polling or can be push based. This interface is technology-dependent and possibly proprietary.

Each search node runs a process that communicates with the associated network element(s) from which it retrieves data in a networked system. A database function dynamically maps that data into the information model for network search and updates the local search database.

Search functions, invoked from the management plane through query invocation, are executed as distributed algorithms on the graph of search nodes. During the execution of a query on a search node, the local search database is accessed, the matching of the local query against stored indexes is performed, and the local search result is possibly aggregated with results from other nodes.

## 7.4 Object Model

We consider physical and logical entities in a networked system, such as routers, servers, IP flows, virtual machines, etc., as objects in a search space (or object space). We associate an object in the space with each of these entities. An object is modeled as a bag of attribute-value pairs, containing configuration and operational information. An object is named and typed, and, hence, has at least two attribute-value pairs. Figure 7.2 shows two examples, an IP flow object with information available on a router, and a virtual machine object with data from a server.

We name an object using an Uniform Resource Name (URN) [145]. Such a name is a unique, location-independent and expressive identifier. (We choose URNs over Uniform Resource Locators (URLs) [44], because they are more persistent, and we do not consider oblivious identifiers, as used in [146], since they are not sufficiently expressive.) The top attribute-value pair in the objects shown in Figure 7.2 are URNs.
We introduce a relation between objects that links together objects that share attribute-value pairs. The relation will allow us to find information that belongs to a certain context in a networked system. For instance, it will enable us to trace an IP flow passing through the nodes of a network, or to search for the servers in a cluster that run applications belonging to a certain customer.

Consider objects \( a, b \) in a search space \( O \). We say \( a \) is directly linked to \( b \), denoted by \( l(a, b) \), if \( a \) and \( b \) share an attribute-value pair. Obviously, \( l(\cdot, \cdot) \) is reflexive and symmetric, but not transitive. Note that the same relation \( l(\cdot, \cdot) \), with the same properties, can be defined on subsets \( A, B \subset O \). We say \( a \in O \) is linked to \( b \in O \), denoted by \( l^*(a, b) \), if

\[
l^*(a, b) := \begin{cases} l(a, b), & \text{or} \\ \exists c \in O : l^*(a, c) \land l^*(c, b) \end{cases}
\]

The relation \( l^*(\cdot, \cdot) \) is reflexive, symmetric, and transitive, by construction, and, therefore, an equivalence relation. Similarly, as above, one can define \( l^*(\cdot, \cdot) \) on subsets of \( O \). It is often useful to compute the closure of a subset \( A \) under \( l^*(\cdot, \cdot) \). For instance, all information associated with a video service can be found in the closure of the set of flow objects related to the service.

The above described model is simpler and coarser than the information models traditionally used in network management, such as, SMIv2 [5], GDMO [147], CIM [148], and YANG [6], but it is also less expressive. We believe that our model is better suited for network search, as one can formulate queries with minimal knowledge about information structure. Furthermore, one can easily populate our model with data from available sources in a network system, at the price of potentially losing structural information.
7.5 Query Language

A query on a search space \( O \) returns a subset of information in this space. We describe the query language in BNF notation as follows:

**Basic:**

\[
q \rightarrow t \mid q \land q \mid q \lor q \\
t \rightarrow a \mid v \mid a \ op \ v
\]

(7.1)

\[
op \rightarrow = \mid < \mid \approx \mid \cdots
\]

(7.3)

**Link:**

\[
q \rightarrow \lambda A \mid \lambda^* A
\]

(7.4)

**Projection:**

\[
p \rightarrow q \mid \pi q
\]

(7.5)

**Aggregation:**

\[
r \rightarrow p \mid \alpha p
\]

(7.6)

First, we discuss queries \( q \) based on rules (7.1), (7.2), (7.3), which return a set of objects. Rule (7.2) defines how a query is made up of tokens \( t \). \( a \) stands for an attribute name, such as \( \text{load} \), \( v \) for a value, such as 0.7, and \( op \) for a relational operator. Here are some examples of queries based on rules (7.1), (7.2), (7.3): \( \text{load} \), \( \text{load} > 0.7 \), \( \text{server} \land \text{load} > 0.7 \), \( \text{server} \lor \text{router} \), (\( \text{server} \lor \text{router} \)) \( \land \text{load}=0.7 \).

Each token \( t \) expresses a condition on an attribute-value pair. During query processing, the token is matched against all objects in the search space—more precisely, against all attribute-value pairs of objects in the search space. If the match is successful, then the object is included in the query result. For example, the token \( \text{server} \) returns all objects that contain the attribute name \( \text{server} \). The token \( \text{load} \geq 0.7 \) returns all objects that include an attribute named \( \text{load} \) with a value larger than 0.7.

Note that the match of a token to an attribute-value pair does not have to be exact, but can be approximate, for an object to be included in the query result. Approximate matching applies to value as well as to the attribute name. We consider approximate matching as an important characteristic of network search. The degree to which a token matches an attribute-value pair can be reflected in the ranking of the object in the query result. This issue is part of our future work.

The query \( q_1 \lor q_2 \) returns the union of the results of sub-queries \( q_1 \) and \( q_2 \). Likewise, \( q_1 \land q_2 \) returns the intersection of sub-queries \( q_1 \) and \( q_2 \). We give an example from a datacenter that offers Infrastructure-as-a-Service: “find servers with at least 12 CPU cores and that have load lower than 20 percent,” which can be expressed as

\[
\text{server} \land \text{cpuCores} > 11 \land \text{load} < 0.2
\]

A special case in matching occurs when the token contains a name. We allow a substring of the URN to be given as a name, for example, John instead of urn:ns:customer:John:Doe. If the substring matches the value of an object name, then the object is returned. This way, query processing performs name resolution.

Rule (7.4) describes link queries, whereby \( A \) denotes a set of objects, \( \lambda \) denotes the operator for direct linking, and \( \lambda^* \) denotes the operator for linking. In case of operator \( \lambda \), the above query returns the directly linked objects of \( A \). In case of \( \lambda^* \), it returns the closure of \( A \) with respect to \( l^*(\cdot, \cdot) \), which means all objects \( o \in O \)
that are linked to objects in \( A \). The following query computes the closure of an object and returns objects of a specific type from that closure: “find servers that run processes of customer John”:

\[
\text{server} \land \lambda^* \text{John}
\]

Rule (7.5) introduces a projection operator \( \pi_{a_1, \ldots, a_n} \), which is applied to a set of objects and returns those attribute-value pairs whose attribute names \( a_1 \cdots a_n \) are specified. This operator reduces the amount of information returned by a query, by giving back a subset of attributes for a particular set of objects, instead of all attributes of these objects. For instance, the query “find servers with at least 12 CPU cores” can be written as

\[
\pi_{\text{name}}(\text{server} \land \text{cpuCores} > 12)
\]

Rule (7.6) describes an aggregation operator \( \alpha_{f, a} \), which takes as input a set of objects \( B \) and returns the value of the aggregation function \( f \) applied to the values of attribute \( a \) in all objects in \( B \); i.e., \( \alpha_{f, a}(B) = f(b_a | b \in B) \). Typical aggregation functions are \( \text{sum}, \text{count}, \text{max} \) and other statistical functions, although non-numerical functions can be used. This operator computes an aggregated value of a set of objects, rather than returning the objects themselves. The query “find the virtual machine with the highest CPU cores on server X” can be expressed as

\[
\alpha_{\text{max, cpuCores}} (\text{VM} \land *\text{server}*X)
\]

We briefly compare our object model and query language with that of the relational model with its standard operators selection, projection, aggregation and semi-join [149]. If we restrict our object model in such a way that objects of the same type have a predefined set of attributes only, and tokens are of the form \( a \ op v \) only, then our model becomes as expressive as the relational model, in the following sense. An object in our model corresponds to a tuple in the relational model and vice-versa; objects of the same type correspond to a relation and vice-versa. Now, it is straightforward to show that queries on our model produce results that corresponds to those results the operators on the relational model produce. For example, a basic query made up of rules (7.1), (7.2), (7.3) corresponds to a selection operation (possibly with set union or intersection), a (direct) link query expressed by rule (7.6) corresponds to a semi-join operation, etc. To see the latter, consider two set of objects \( A_1 \) and \( A_2 \) and their relations \( R_1 \) and \( R_2 \). It is then straightforward to see that the query \( A_1 \land (\lambda A_2) \) in our model produces the result that corresponds to the relational query \( R_1 \bowtie R_2 \).

While our model is as expressive as the relational model, it is more general. First, it allows for objects of the same type to have different attributes. This generalization allows us to capture the heterogeneity of network entities. For instance, network functions like firewalls come in different varieties and, therefore, need to be described using different attribute sets. Second, our model replaces the tuple identifier in the relational model, i.e., the key attributes, with a single attribute, namely, the object name. Third, in our model, an attribute name or an attribute value, can be given as a query, which is not possible in the relational model. This allows us to invoke
queries without explicit knowledge of the object structures, and it enables us to give values without providing the corresponding attribute names as well. Lastly, the link operator in our model does not have a corresponding operator in the relational model (although the direct link operator has, as discussed above).

### 7.6 Distributed Query Processing

Our approach to process network queries makes use of the echo protocol, a tree-based protocol suitable for distributed polling [13] [150]. It is based on an algorithm first described by Segall [151]. The execution of the echo protocol can be understood as the subsequent expansion and contraction of a wave on a network graph. The execution starts and terminates on an initiating node of the graph, also called the root (node). The wave expands through explorer messages, which nodes forward to their respective neighbors. During the expansion phase, local operations are triggered on the nodes after receiving an explorer. The results of these local operations are collected in echo messages, when the wave contracts, so that the aggregated result of the global operation becomes available at the root node. During the expansion phase, the protocol constructs a spanning tree on the network graph for the purpose of collecting and aggregating the partial results during the contraction phase.

The echo protocol executes on the network graph of the search plane (Figure 7.1). The protocol can be started on any search node once a query $q$ has been received. First, the query is disseminated by explorer messages to every node and executed as local operation against the local database $D$. The results of the local operations are sent, by echo messages, on the spanning tree from child nodes to parent nodes, where the partial results are aggregated. (Note that the term aggregation here refers to the processing of the partial query results, not to a possible aggregation operator in the query $q$.) Figure 7.3 shows a sample spanning tree created by the echo protocol on nodes $n_1, \ldots, n_6$ with root $n_1$. It further shows the message exchange between nodes. Each node shows the local database $D$ containing the objects with information from that node, together with the variable $result$, which contains the (partial) result of the query $q$.

The definition of the local operation, the aggregation operation of the query result, and the current local state of the query collection, are modeled in an object, called the *aggregator object* of the echo protocol (see Figure 7.4). The aggregator in the figure contains the code to process a query $q$ that contains neither $\lambda$ nor $\lambda^*$ operator. Line 2 defines the variable $result$, which is either a set of objects or, it contains an aggregated value of this information in case the query contains an aggregation function. Lines 3-4 defines the local function and lines 5-9 defines the procedure how partial results are aggregated. If the query $q$ includes an aggregation operator $\alpha_{f,a}$, then variable $result$ is updated using the aggregation function $f_a$, otherwise by computing the union of the partial results.

We describe now the processing of queries $q$ that contain link operators. First, if $q$ is of the form $\lambda A$, whereby $A$ is a set of objects, then the local function in the aggregator object can be written as $A \cup \{o \in D - A \mid \exists a \in A : l(a,o)\}$. Second,
Figure 7.3: Distributed query processing: The echo protocol creates a spanning tree in the search plane. Each node contains the local database $D$. The variable $result$ contains the partial result of a query $q$. Some of the explorer (EXP) and echo (ECHO) messages are shown.

1: aggregator object processQuery( )
2: var: result:QueryResult;
3: procedure local( )
4:   result := $q(D)$;
5: procedure aggregate(childResult:QueryResult)
6:   if $q$ contains $\alpha_{f,a}$ then
7:     result := $f_a(result, childResult)$;
8: else
9:     result := result $\cup$ childResult;

Figure 7.4: Aggregator for processing a query without link operators.

if $q$ is of the form $\lambda q'$, then processing $q$ requires two executions of echo; during the first execution, $q'$ is computed, and the second execution computes the directly linked objects. Finally, in case $q$ is of the form $\lambda^* q'$, at least two executions of echo are required, whereby, first, $q$ is processed, then $\lambda q'$, $\lambda(\lambda q')$, $\lambda(\lambda(\lambda q'))$, etc. The process stops when the output set does not grow from one execution to the next.

The proof that the above described distributed processing technique is correct draws from the fact that the global database, which contains all objects, is horizontally fragmented; each object is stored in exactly one local database. The performance characteristics of distributed query processing is based on the performance properties of the echo protocol [13]. For instance, the execution time of a query
grows proportionally with the height of the spanning tree, which is upper bounded
by the diameter of the network graph. The protocol overhead is evenly distributed
on the network graph, as two messages traverse each link during the execution of
echo. Lastly, the number of messages each search node processes is upper bounded
by the degree of the network graph. While the above properties suggests that the
presented generic approach to network search is a scalable solution, major challenges
remain to engineer a large-scale network search system. See Section 7.8.

7.7 Use Cases

Use Case: Networking

The delivery of complex networking services involves increasing numbers of entities
with numerous interdependencies, many of which are only temporary and fleeting
in nature. With the advent of cloud services and virtualization, many of the entities
themselves are increasingly short-lived, as layers of virtualized entities build on each
other. In addition, many decisions that affect end-to-end service characteristics
depend on very dynamic decisions, such as performance routing in which the path
of individual packets in a media stream depends on short-lived fluctuations in load
and service levels. As a result, management paradigms need to increasingly shift
away from planning and predicting, towards monitoring and tracing (packets in a
network, dependencies between resources, policy decisions imposed on a flow, etc.),
in order to understand what is precisely going on, should it be required.

However, tracing, while tremendously important, can be a challenge. This is
where network search can provide some distinct advantages. Consider the case in
which an operator is concerned about an interactive video session whose service level
is deteriorating. The operator has no way of knowing which particular systems or
links the stream is traversing and might contribute to the problem. Using network
search, the operator can issue a network query to retrieve a list of all systems in
the network with a flow record which contain the source and destination systems
as part of their flow keys, along with the flow records and the interface statistics
of the incoming and outgoing interfaces. The search provides the operator with
data from every node in the network that participates in the flow. From this, an
application can perform further analysis and stitch together which path the flow
takes, whether multiple routes (perhaps due to load balancing considerations) are
taken, where packets are dropped, etc.

Now consider the same scenario but involving networking boundaries at which
NAT (Network Address Translation) is performed. For example, the session might
involve two endpoints in two different enterprises and traverse a public service
provider network. In that case, network address translation occurs at the bor-
der between enterprise A and the service provider, and again at the border between
the service provider and enterprise B. As a result, the same session is associated
with different flow keys in each of these domains. Hence, network search cannot
be applied naively in the same query being issued to every device in either of the
involved networks. Instead, the query itself needs to be locally adapted according to
the network address translations at the different network boundaries—the network
search needs to be NAT-aware. For this purpose, we introduce NAT-aware search
nodes at nodes that involve a NAT function. A NAT-aware search node has access
to the same NAT database as the NAT function itself. Whenever a search query or a
search result involves an IP address in the NAT database, the NAT transformation
is applied as the query traverses the network boundary.

Search on a Cloud Infrastructure

In order to experiment with network search concepts, we have instrumented a cloud
Infrastructure-as-a-Service (IaaS) platform in our laboratory with network search
functions. The platform contains nine high-performance servers, interconnected
by Gigabit Ethernet, and runs the OpenStack cloud management software. (See
[152] for details.) The components of the network search system include search
nodes—each server on the testbed runs such a node—and managers in the manage-
ment plane that run in a management station. Each search node contains a local
database based on MongoDB [153] that maintains the network objects. Currently
the database has four types of objects, namely, server, virtual machine (VM), appli-
cation, and customer. A data sensing component reads system files, such as ‘/proc’
[154], libvirt configuration [155] etc., and populates and periodically updates the
objects in the local database at a rate corresponding to their respective lifetime. A
distributed query processing component implements the protocol outlined in Section
7.6. The manager component offers two types of interfaces for accessing network
search functionality: a simple line console and a graphical, menu supported inter-
face that allows to compose queries and browse the output in various ways. Due to
lack of space, details about this implementation will be reported elsewhere.

We have used the network search system on the OpenStack platform for conduct-
ing a range of exploratory experiments. The platform can be loaded by external load
generators that were developed for evaluating performance management solutions
for OpenStack [152]. The produced load has a time-varying pattern of several types
of applications running in virtual machines of different configurations and lifetimes.

Here are some of the experiments we have performed. First, we inquire about
the load on a server cluster, which is given by a range of IP address: $\alpha_{\text{sum, load}} (192.168.212.*)$. Given the case that the load is unexpectedly high, we want to
find out which applications are running on this cluster: $\pi \text{name}(\text{application} \land \lambda^* 192.168.212.*)$. Finally, we want to identify the customers for which these ap-
plications are executed: $\pi \text{name}(\text{customer} \land \lambda \text{app}_1 \lor \cdots \lor \text{app}_n)$. In a further set of
experiments, we study the behavior of the virtual machines on the platform under
an adaptive placement policy. First, we are interested in learning the distribution
of the uptimes of the virtual machines. The query $\pi_{\text{uptime}}(\text{VM})$ provides the up-
time of the active virtual machines, out of which the distribution can be computed.
(If a distribution aggregator is implemented, then the distribution can be directly
computed as part of the query). Second, we want to study the movement of virtual
machines that belongs to a specific application. We can achieve this by periodically
issuing the query $\pi_{\text{name, server}}(\text{VM} \land \text{app}_x)$.

We ran a series of performance tests on the search system, the details of which
can be found in [156]. For some experiments, the search space was populated with
2000-3000 objects on each server, which took up some 1-1.5 MB of disk space. In
the current configuration, objects are refreshed once per second via polling, which
creates a CPU load of about 2 percent on each server, for local databases with 2000
objects. Global search queries, which include projection and aggregation operators,
execute in approximately 20 milliseconds, while link queries can take considerably
more time. An analysis of the performance measurements suggests that there is a
room for performance improvements in the implementation of the current prototype.

7.8 Discussion

The contribution of this paper centers around a simple query language for search
in networked systems. Queries are based on a model where objects are represented
as a set of attribute-value pairs. We propose a method for distributed execution of
search queries in a networked system in which nodes maintain objects that contain
configuration and operational information. We argue why the proposed method
provides the correct result for a network query. Two use cases further motivate the
paradigm of network search and demonstrate that the introduced language is useful.
Our implementation gives evidence that the design can be implemented (even if the
testbed is of limited size).

Similar to the case of web search, the simplicity of our query language has the
drawback that, often, the information we are interested in cannot be expressed in
a sufficiently precise manner, and, therefore, the query result needs interpretation.
For instance, the query “search for servers that run processes of customer John”
cannot be directly expressed in our query language, but only through attribute
names, values, and object links, for example, through $server \land \lambda^* \text{name} = ^*\text{John}$.
This query returns objects with attribute name (or value) server that are linked to
objects whose names end with John. The query result needs to be interpreted and,
hopefully, contains the information we searched for in the first place.

An argument can be made that search queries can be implemented as specialized
protocols in a networked system, and, therefore, a generic search system is not
needed (see Section 7.7). However, we believe that a network search system enables
new functionality to be added on-demand, or it allows for network applications to
dynamically adapt their information demand.

In future work, we plan to further develop the paradigm of network search. Here
are some of our priorities. While the contribution in this paper focuses on database
aspects of network search, network search includes concepts that go beyond database
functionality, most importantly, approximate matching of attributes and ranking of
search results (see Section 7.1). We envision, for instance, that ranking takes into
account the freshness of the data, the locality of the query invocation, and the
number of tokens in a query that matches a particular object, and that the ranking
process is realized as a distributed aggregation function.

Second, we plan on improving the scalability of distributed query processing
for network search. While this paper describes a distributed method for query
processing, each query still invokes an operation on every search node, which is
expensive in a large system. We are considering several approaches to reduce the
footprint of a query. As pointed out in Section 7.2, domain knowledge can be used to guide the search process and thus reduce the search space. Alternatively, an index structure can be developed to reduce the number of nodes that are involved in processing a query. Link queries require special attention, since each such query involves several executions of echo. Possible heuristics for reducing the overhead include restricting the search to those links with the number of intermediate objects below a given bound, and limiting the subsequent executions of echo to those nodes that produced a non-empty query result during the previous execution.

Additionally, work is needed for the development of efficient local databases, the population of local database with available data sources, the development of concepts regarding the privacy and security of local data, as well as a framework for search in a multi-domain environment.
Chapter 8

Scalable Matching and Ranking for Network Search

Misbah Uddin†, Rolf Stadler†, Alexander Clemm‡
† KTH, Royal Institute of Technology, Sweden. Email: {ahmmud,stadler}@kth.se
‡ Cisco System Inc., San Jose, USA. Email: {alex}@cisco.com

Abstract

Network search makes operational data available in real-time to management applications. In contrast to traditional monitoring, neither the data location nor the data format needs to be known to the invoking process, which simplifies application development, but requires an efficient search plane inside the managed system. The search plane is realized as a network of search nodes that process search queries in a distributed fashion. This paper introduces matching and ranking for network search queries. We are proposing a semantic for matching and ranking, which is configurable to support different types of management applications—from exact matching for database-style queries to loose, approximate matching, which is appropriate for exploratory purposes. We describe an echo protocol for efficient distributed query processing that supports matching and ranking. Further, we present the design of a search node, which maintains a real-time database of operational information and allows for parallel processing of search queries. A prototype implementation on a cloud testbed shows that the network search system, on a 9-node cluster with 24 core servers, executes 200 global search queries/sec with the 75th percentile latency below 100 milliseconds and with a CPU utilization below 5%. The performance measurements, together with our design, suggest that a system of 100’000 servers processing the same load would exhibit the same overhead per server and a query latency of below 1 sec.

8.1 Introduction

Network search is a paradigm that stresses an information-centric view of network management [1, 2]. Its main elements are real-time access to operational and configuration data, a weakly structured data model, and a scalable search function that
executes inside the managed system. Network search is specially suited for large-scale dynamic networks and networked systems, and it enables novel management functionality, such as “network googling” for root cause analysis of faults, dynamic asset tracking, and real-time network analytics.

The key difference between network search and traditional web search is that the former relates to search for real-time information inside a networked system, while the latter centers around search for non real-time data in an offline index database. While many concepts of web search are applicable to network search, the challenge is to realize them under the stringent requirements of network search.

In earlier work, we proposed a design of a network search system, including a protocol for scalable query processing. In this paper, we investigate the semantics of network search queries with respect to query matching and result ranking. Elements of our proposed solution are based on results from information retrieval and web search; they make use of the extended boolean retrieval model [15], as well as connectivity metrics and attribute frequency of information objects.

We believe that a network search system should offer, for the same query language, a range of matching and ranking semantics, in order to support different types of management applications. For some applications, database-style exact matching is appropriate, where each data item either fully matches a query or does not match at all; for others, loose, approximate matching is more suitable, where data items match a query to different degrees. The same applies to ranking: for some applications, objects with more attributes of a given type are more relevant—and thus ranked higher—than objects with fewer such attributes; for other applications, objects with recently updated attributes are more relevant than those with more stale information, etc.

Consider a search query that includes a sequence of IP addresses. For an asset tracking application, for instance, matching is best performed in such a way that exactly the objects with those IP addresses are retrieved, and their ranking order is irrelevant to the application. In contrast, consider a network-security administrator who receives a set of IP addresses from an intrusion detection system and runs a network search query with these addresses. In this case, the query semantics is such that it matches objects representing devices, flows, applications, etc., that are associated with these addresses. In the query result, highly connected devices rank high, as well as current flows.

With this paper, we make the following contributions. We introduce a semantic for matching and ranking that is tailored to network search. We show how the semantic can be realized in a distributed query processing protocol, which has sound scaling properties. Both the matching and ranking functions are configurable. The possible configurations range from supporting exact matching, which allows database-style retrieval, to loose, approximate matching, which is similar to the matching semantics of the vector space model in information retrieval [18], and which supports exploration of the information space. Configuration parameters can be given at query invocation time, and our design allows a network search system to concurrently process queries with different matching and ranking semantics. The paper further presents the design of a search node supporting multicore hardware. Lastly, we report performance figures from a prototype implementation on a cloud
testbed, which demonstrates the feasibility of engineering a high performance network search system.

8.2 Related Work

Most of the research in matching and ranking has been performed in the context of web search or search related to extensions of traditional web technologies. Many concepts that have been applied in web search and possibly refined later are based on earlier work in information retrieval (IR). IR does not generally distinguish between matching and ranking. Given a search term, an IR system produces a set of objects, each associated with a matching score. This score is often based on two metrics: the term frequency $tf$ and the inverse document frequency $idf$ [16]. Two examples of IR models that are relevant to this paper are the vector space model [18] and the extended boolean retrieval model [15]. A brief discussion of IR concepts with respect to network search can be found in [1].

Traditional web search is performed on static web content and uses primarily link analysis for ranking, for example, PageRank [27] and, HITS [28]. Current web search engines, which search static as well as dynamic content, use hundreds of metrics to compute ranking scores; in addition to page ranks, these metrics include a range of page usage statistics, matching scores from different IR models, freshness of data, etc. [29]. The weights of these metrics are generally determined using machine learning techniques applied to query logs [30, 31, 32]. A widely used matching scheme in web search is BM25F, which is an extension of the popular IR model Okapi BM25 [23]. It is based on partitioning information on a web page into different fields, which are matched according to their importance.

With the evolution of Web 2.0, new functionality has been introduced, which gives rise to new metrics for ranking. For instance, in social tagging systems, metrics based on tags for pages and links influence the ranking score [39, 157]; in live search systems, the freshness of the data influences the score [36, 34]. Recently, web-based frameworks have been developed that provide access to specific types of information. For instance, the Web-of-Things framework, which supports Internet-of-Things technology, includes search functions that match against numerical values [50, 46]. Second, the Web-of-Data framework makes available massive data sets in form of graphs. Term matching is performed against attributes associated with graph nodes [58, 59, 60, 56].

8.3 A Distributed Architecture for Network Search

Figure 8.1 shows an architecture for a network search system, which we introduced in [1]. Its key element is the search plane, which conceptualizes the network search functionality. This plane contains a network of search nodes, which have processing and storage capacities. A search node can communicate with a set of neighbors, which are identified through links of the network graph. The design of this plane supports searching in a distributed and parallel fashion. A search node can be realized in various ways: it can be part of the management infrastructure outside
the managed system, it can be run as a standalone network appliance, or it can be integrated into a network element using a variety of technologies. Our current prototype implements the third option.

The bottom plane in Figure 8.1 represents the physical network that is subject to search. Each network element is associated with a search node, which maintains (or has access to) configuration and operational data from that network element. This data is modeled as a set of objects, whose structure is described in Section 8.4. Note that the figure shows the simplest form of association between a network element and a search node; it is possible that a search node maintains data from several physical devices, or, alternatively, a device updates data on several search nodes. The top of Figure 8.1 shows the management plane, which includes the systems and servers running processes for network supervision and management.

There are three important interfaces in this architecture. The first is the search interface, which supports the query language discussed in Section 8.4. We envision that every search node is an access point for search queries. The second interface defines the interaction between a search node and a network element, which can be realized through polling or can be push based. This interface is technology-dependent and possibly proprietary. The third interface is the peer interface between the neighboring search nodes. It enables node to interact for distributed processing of search queries.

Each search node runs a process that communicates with the associated network element(s) from which it retrieves data in a networked system. A database function dynamically maps that data into the information model for network search and updates the local search database.

Search functions, invoked from the management plane through query invocation,
are executed as distributed algorithms on the graph of search nodes. During the execution of a query on a search node, the local search database is accessed, the matching of the local query against stored indexes is performed, and the local search result is possibly aggregated with results from other nodes.

### 8.4 A Semantic for Matching and Ranking of Network Search Queries

Following [158], our model for network search includes a four-tuple \( \langle O, Q, M, R \rangle \). Here, \( O \) is the set of objects that represents the object space; \( Q \) is the set of syntactically correct queries on this space; \( M \) is the matching function, which determines, for a query, the set of objects in the query result; \( R \) is a ranking function, which orders the objects in the query result. In the following, we review the object model and the query language. Then, we introduce the matching and ranking functions for network search.

#### The object model

In [2], we presented an object model for network search. Objects in this model can represent physical and logical entities in a networked system, such as routers, servers, IP flows, virtual machines, etc. An object is modeled as a bag of attribute-value pairs—for short, attributes—whereby an attribute can contain configuration or operational information. An object has a unique name and a type, both of which are modeled as attributes. Figure 8.2 shows three sample objects. We say two objects are linked if they share an attribute.

<table>
<thead>
<tr>
<th>object-name : cloud-8</th>
<th>object-name : instance-7</th>
<th>object-name : instance-18</th>
</tr>
</thead>
<tbody>
<tr>
<td>object-type : server</td>
<td>object-type : VM</td>
<td>object-type : VM</td>
</tr>
<tr>
<td>cpu-core : 12</td>
<td>cpu-core : 2</td>
<td>cpu-core : 1</td>
</tr>
<tr>
<td>memory : 32 GB</td>
<td>memory : 4 GB</td>
<td>memory : 2 GB</td>
</tr>
<tr>
<td>IP-address : 12.1.1.1</td>
<td>IP-address : 12.1.1.5</td>
<td>IP-address : 12.1.1.9</td>
</tr>
<tr>
<td>IP-address : 12.3.1.3</td>
<td>server : cloud-8</td>
<td>server : cloud-7</td>
</tr>
<tr>
<td>IP-address : 17.3.1.3</td>
<td>load : 0.5</td>
<td>load : 0.4</td>
</tr>
<tr>
<td>load : 0.85</td>
<td>customer : john:watson</td>
<td>customer : john:watson</td>
</tr>
<tr>
<td>uptime : 113.47</td>
<td>uptime : 350.47</td>
<td>uptime : 650.47</td>
</tr>
<tr>
<td>OS : ubuntu</td>
<td>hypervisor : kvm</td>
<td>hypervisor : kvm</td>
</tr>
</tbody>
</table>

Figure 8.2: Sample objects in a search space. (a) an object that represents a server in a cluster, (b)–(c): two objects that represent virtual machines in servers.
CHAPTER 8. SCALABLE MATCHING AND RANKING FOR NETWORK SEARCH

The query language

In [2], we presented a query language for network search, the core of which is given in BNF notation:

\[
q \rightarrow \ t \mid q \land q \mid q \lor q
\]

\[
t \rightarrow \ a \mid v \mid a \ op \ v
\]

\[
op \rightarrow \ = \mid < \mid \cdots
\]

Rule (9.2) states that a token in this language is either an attribute name \( a \), a value \( v \), or an attribute-value pair \( a \ OP \ v \). According to rule (9.1), a query can be a single token, or it can be constructed using tokens and boolean operators. We leave out here the discussion of the projection, aggregation and link operators of the language, because they are not essential to the discussion in this paper.

A semantic for matching search queries

In the simplest case, a matching function \( M \) maps a query and an object onto a boolean value, i.e., \( M : Q \times O \rightarrow \{0, 1\} \). An object either matches a given query, in which case it is included in the result set of the query, or it does not match, in which case it is not part of the query result. We say that \( M \) is an exact matching function, if it maps to a boolean value. Such matching functions are generally used in databases. Also, the semantics of the query language we introduced in [2] is based on exact matching.

Web search does not uses exact matching, but approximate matching. In this case, the matching function \( M \) maps a query and an object onto a matching score in the interval \( 0–1 \), i.e., \( M : Q \times O \rightarrow [0, 1] \). If \( M \) returns 0, the object is not included in the result set of the query; otherwise, it is included, and the value indicates the relevance of the object for the query: the higher the value the better the match.

Network search is similar to web search in the sense that search is often used to explore an information space, and the simple syntax of the query language does not always allow to express the intention of the invoker. Therefore, approximate matching should be supported in network search. Note also that exact matching is a special case of approximate matching.

In order to define the matching function \( M \) for approximate matching, we apply the extended boolean retrieval model, a popular IR model which was proposed by Gerard Salton et al. in 1983 [15]. The query language of this model is close to the one we are using for network search. An important part of the model is a similarity function, which measures the similarity between a query and an object, and which is analogous to the matching function \( M \) in the network search model.

Our function \( M \) uses two basic metrics from information retrieval, the term frequency \( tf \) and the inverse document frequency \( idf \). In the network search model, these metrics relate to tokens of the query language. This means that the term frequency expresses the frequency of occurrences of an attribute name, an attribute value and an attribute in an object, and the inverse document frequency indicates the inverse of the number of occurrences of attribute names, attribute values and attributes in the object space. To give a specific example, consider the query with
8.4. A SEMANTIC FOR MATCHING AND RANKING OF NETWORK SEARCH QUERIES

the token IP-address. The term frequency for this token and object (a) in Figure 8.2 is the number of occurrences of this attribute normalized by the total number of attributes of the object, i.e., \( tf = 0.27 \). The inverse document frequency for this token relates to the probability that an object has the attribute IP-address. (We give here simplified version of the formulas for \( tf \) and \( idf \) in order to stress the basic ideas.)

For a specific object \( o \), we compute \( M \) for a token \( t \) as

\[
M(t,o) = tf \cdot idf
\]

Following [15], we define the function \( M \) for queries that are constructed out of \( n \) tokens and boolean operators as follows:

\[
M(q_1 \lor \cdots \lor q_n) = \frac{||M(q_1) \cdots M(q_n)||_p}{\sqrt{n}} \tag{8.4}
\]

\[
M(q_1 \land \cdots \land q_n) = 1 - \frac{||1 \cdash M(q_1) \cdash \cdash 1 \cdash M(q_n)||_p}{\sqrt{n}} \tag{8.5}
\]

Equations 8.4 and 8.5 use the \( L_p \) vector norm, also known as P-norm. Choosing \( p = \infty \) results in \( M \) being the matching function of the (conventional) boolean model in IR, i.e., \( M \) performs exact matching [17]; choosing \( p \) in the interval \([1, \infty)\) results in an approximate matching function. Changing \( p \) to a smaller number increases the number of matched objects for a given query. For \( p = 1 \), \( M \) becomes the matching function of the vector space model in IR and performs loose approximate matching [18]. With equations 8.4 and 8.5, the matching function on any boolean expression is well-defined.

In our matching model, all tokens have the same weight. The framework in [15] upon which we base our matching function allows us to include weights for tokens in a straightforward way. We plan to study such an extension as part of our future work. (Models for term weights are important for web search systems.)

A ranking function

A ranking function \( R \) maps the result set of a query \( q \) onto an ordered list. The result set \( O_q \) of query \( q \) includes those objects in the object space whose matching score is positive, i.e., \( Q_q = \{ o \in O | M(q,o) > 0 \} \). The first element of the list is the object that is most relevant to the query, and the relevance decreases with each subsequent list element. In search systems, this list is often truncated after \( k \) elements in order to limit the size of the result set. Note that the matching function \( M \) introduced in the above Section 8.4 is a ranking function. In the following, we will extend \( M \) to include heuristics that are relevant to network search, and we will express the relevance of an object to a query in form of a ranking score.

The first metric we are considering measures the similarity between an object and the query, the second relates to the connectivity of an object within the graph of all objects, whereby links between objects express relationships through joint attributes (see Section 8.4), and the third relies on information freshness. In the search literature, such metrics are also called weights or signals.

First, our similarity metric uses the matching function \( M \) discussed above, extended to accommodate our object model. The matching rule for an object name is extended, so that a token matches a substring in a name, e.g., ‘urn:ns:john:watson’. Also, the contribution to the matching score is higher for the
name and the type attributes than for other attributes in an object, since we consider objects matching a query via name or type more relevant to the query than objects matching the query via other attributes. Second, the ranking score considers the connectivity of an object within the graph of all objects. The intuition behind this metric is that a high connectivity of an object signifies a high importance of this object. In web search, the same idea of measuring connectivity is behind the pagerank or hub score metrics [115]. Third, the ranking score considers the freshness of the information contained in an object: the more recent the information, the higher the score, which means that fresh information is more relevant to a query. Finally, note that other metrics can be considered when computing the ranking score, for instance, metrics relating to location or search history of a query invoker. Such metrics have proven useful in other search systems, and we plan to study them for possible inclusion in our network search system. We compute the ranking function $R$ as a weighted sum of the above metrics.

The matching and ranking functions discussed in this paper compute, for a given query $q$ and object space $O$, the result set and an ordering of objects within the set. The position of an object within this ordering expresses its relevance with respect to the query. It is important to recognize that this relevance is highly dependent upon the management task that uses network search to obtain information from the network. For instance, consider three different management tasks: a human operator who “googles the network” to identify the root cause of a fault, a cloud management application that performs virtual asset tracking, and an anomaly detection application that searches for abnormal patterns in network state information. For the same or similar queries, all these applications may consider different result sets as appropriate and different ordering policies to reflect their respective relevance. For this reason, matching and ranking modules in a network search system must be generic, so that they can be instantiated for specific application purposes. In our current design and system implementation, the matching function can be initialized with different term weights and different values for the $p$ vector norm ($p = \infty$ for exact matching, $p < \infty$ for approximate matching, and the default is $p = 2$). The ranking function can be initialized with different weights for the metrics discussed above. A weight can be set to 0 to ignore a specific metric.

8.5 Distributed Processing of Search Queries

Our approach to process network search queries makes use of the echo protocol, a tree-based protocol suitable for distributed polling [13, 150]. It is based on an algorithm first described by Segall [151]. The echo protocol executes on the network graph of the search plane (Figure 8.1). It can be started on any search node once a query $q$ has been received. First, the query is disseminated to every node and executed against the local database $D$. The results of all local operations are sent along a spanning tree, where the partial results are aggregated. The definition of the local operation, the aggregation operation of the query result, and the current local state of the query collection, are modeled in an object, called the aggregator object of the echo protocol.
8.5. DISTRIBUTED PROCESSING OF SEARCH QUERIES

1: **aggregator object** processQuery( )
2: \[\text{var: qr : dictionary; }\]
3: **procedure** local( )
4: \[\text{qr := \{\}; }\]
5: \[\text{for each } o \in M(q,D) \text{ do }\]
6: \[\text{insert (name(o), o, R(q,o)) into qr; }\]
7: \[\text{qr := top-k(qr); }\]
8: **procedure** aggregate(child-qr: dictionary)
9: \[\text{qr := top-k(merge(qr, child-qr)); }\]

Figure 8.3: Aggregator for processing a query \(q\) on a node with local database \(D\).

Figure 8.3 shows the aggregator object for distributed processing of queries expressed in the language given in Section 8.4. The object contains the functionality for local data retrieval, matching and ranking, as well as for incremental aggregation of the partial results along a spanning tree. The state of the aggregator object is captured in the variable \(qr\), which stores the local search result of query \(q\), potentially also an aggregated search result. \(qr\) is of type dictionary, which contains 3-tuples of the structure (object name, object, ranking score). Each tuple contains an object retrieved from the local database, together with its ranking score relative to query \(q\). The dictionary function \(\text{insert()}\) inserts a tuple into a dictionary, the function \(\text{merge()}\) combines two dictionary into a single one, and the function \(\text{top-k()}\) creates a dictionary with the top \(k\) tuples, according to decreasing ranking score, of a given dictionary. The procedure \(\text{local()}\) in Figure 8.3 initializes the aggregator state, executes the query \(q\) against the local database \(D\), implicitly through invoking the matching function \(M\), storing the query results in \(qr\), which includes invoking the ranking function \(R\), and reducing \(qr\), if needed, to include the top \(k\) objects only. The procedure \(\text{aggregate()}\) aggregates the local state \(qr\) with the state \(child-qr\) from a child node by merging the two dictionaries and reducing the size of the resulting dictionary if applicable. This procedure implements the process of the distributed aggregation by the echo protocol. Figure 8.3 contains the partial pseudocode of the aggregator object. For instance, the processing of operators of the full query language [2], including, the projection, aggregation, and link operators, is missing. An aggregator object that processes these operators is given in [2]. Second, the code in Figure 8.3 does not reflect the fact that the matching function \(M\) and the ranking function \(R\) are parameterized (see Section 8.4). These parameters must be passed together with the query \(q\) to the aggregator object. Third, both \(M\) and \(R\) rely on object metrics, which must be retrieved from the local database. Two of these metrics, \(idf\) for each attribute and \(connectivity\) for each object, are global and are computed using a global aggregation protocol, which runs independent from query processing.

The performance properties of the echo protocol [13], which determines key performance metrics of the query processing scheme, suggests that the presented generic approach to network search is a scalable solution. For instance, the execution
time of a query grows proportionally with the height of the spanning tree, which is upper bounded by the diameter of the network graph. The protocol overhead is evenly distributed on the network graph, as two messages traverse each link during the execution of echo. Lastly, the number of messages each search node processes is upper bounded by the degree of the network graph.

8.6 Design and Implementation of a Search Node

The search node is the key architectural component of a network search system (Figure 8.4). All search nodes are identical in functionality and co-operatively provide the network search service. A search node has an interface to the management plane, which provides an access point for network search, it executes distributed query processing (echo protocol, local data access, matching, ranking, and aggregation), it maintains a real-time database with network objects, and it includes a sensor subsystem that populates this database. The performance goals for designing a search node are low latency for search queries, for supporting real-time search requirements; low computational overhead for query processing, since search nodes may be hosted by service devices; high throughput of search queries, since we expect a high number of concurrent queries in a large system; support for a large number of network objects (larger than 10,000), to make available to network search an extensive set of operational data.

Figure 8.4 shows the main components of a search node and their interactions. A search node has three interfaces: to the management plane, to peer nodes of the search plane, and to associated devices (possibly internal to a device) of a networked system. The component on top of the figure is dedicated to query processing. It interacts with the management plane, peer nodes and the local database. The component on the bottom includes the sensing functionality that updates the database.

The component in the middle of the figure contains the database module. Our design calls for a document oriented NoSQL database with object-level access [159]. Such a database allows us to implement our object model in straightforward way, and it supports the processing of search queries by providing the basic functionalities for token matching through the database query interface. The object database is complemented with an index structure with entries of the form (key, object id, matching metric, ranking metric). The key field refers to a token in the query language, object id is a pointer to the object in the database, and the matching and ranking metrics contain information needed by the matching and ranking functions for query processing. The index significantly increases the performance of token matching, at the expense of maintaining it.

Distributed query processing is achieved by local processing and exchanging of messages between search nodes. Figure 8.5 shows our design for distributed query processing. Each circle (in the bottom of Figure 8.5) represents a thread that executes asynchronously. The distributed design allows us, on a multicore hardware, to achieve a higher throughput for processing search queries and a lower query latency compared to a straightforward single-thread design. On top of the figure, we find buffers for incoming and outgoing messages, two buffers for each peer node.
or process in the management plane. A message router retrieves the messages from the message in-buffers and places them in the input queues of the query processors. Messages that relate to the invocation of the same query, i.e., which contain the same invocation identifier, are assigned to the same processor. A query processor retrieves a message from its input queue and executes a query processing step. Such a step includes updating the state of the query invocation, for instance, updating the states of the echo protocol and its aggregator, and executing the procedures in the aggregator, for instance the procedure $local()$, which accesses the local database. As part of the query processing step, one or more messages are generated, which the query processor inserts into the output queue. Another message router places messages from this queue into the appropriate message out-buffers. The optimal number of query processors in a system configuration is dependent on the hardware platform of the search node. In addition, it depends on the database capacity and the amount of CPU resources that can be devoted to network search.

We implemented the design of the search node on a multicore architecture. All components, except the database system, are written in Python. We use the multiprocessing package for threading on a multicore hardware. The database component is based on MongoDB [153], a popular opensource database system. MongoDB is a document-oriented NoSQL system that realizes a persistent database. (It includes supports for distributed databases, which we are not using.) We chose a document-oriented database system over a key-value store package, because of the object-level abstractions that a document-oriented database supports. MongoDB allows us to implement our object model in a straightforward manner, supports basic token matching, and exhibits good performance for read and write operations, compared to other document-oriented databases. While MongoDB is not an ideal database system for our purposes—we would prefer an in-memory database with attribute-level locking—we seem to us currently the best choice available. Our current prototype has 5000 objects per search node. The query processing component runs five threads on four cores (message routers share a core), and the database component runs on a single core.

The above design of a search node reflects our performance goals. The purpose of maintaining an index structure is to increase query throughput and lower query latency. The distributed design of local query processing has the same objectives. It also allows us to control the computational overhead of query processing by choosing the appropriate number of cores for this task.

8.7 Evaluation of a Network Search Prototype on a Cloud Testbed

We have instrumented the servers of a cloud platform for network search. The platform includes nine high-performance servers, interconnected by Gigabit Ethernet, and runs the OpenStack cloud management software. (See [152] for details.) Each server includes a search node. The real-time database on a search node has currently four types of objects, namely, server, virtual machine (VM), application, and customer. There are approximately 5000 objects per search node. A data sens-
Figure 8.4: Architecture of a search node

We produce synthetic load for the performance measurements as follows. The query load consists of global search queries with 2 to 5 tokens each. Tokens are either attribute names, values or attribute-value pairs, with equal probability. The values for the tokens are chosen uniformly at random from the global object space. The number of tokens per query is 2 to 5 with equal probability. Queries are injected using a Poisson process and are sent with equal probability to a search node on the testbed. Note that the global query load is the same as the local query load on a search node, since each query is executed on all servers (see Section 8.5). During an experiment, a search node processes a mix of 75 percent global queries and 25 percent local updates. The Local update load consists of insert object, delete object and refresh attribute operations, with equal probability. The objects and...
8.7. EVALUATION OF A NETWORK SEARCH PROTOTYPE ON A CLOUD TESTBED

During an experiment, we measure two metrics: first, the latency of each global query, measured from the time the management station sends out a request until it receives the response; second, the CPU utilization on each server that runs a search node.

Before measurements are performed, the real-time databases on the search nodes are initialized using a script and synthetic data. For each run of an experiment, we inject query/update load at specific rates, wait until the system is in steady state, generally around 10 seconds, and take measurements, over a period of some 120 seconds. The query load for the experiments ranges from 25 queries/sec to 700 queries/sec.

We report on three sets of experiments. First, we investigate the latency of global queries in function of the query load. Figure 8.6 shows the measurement results of runs with query loads ranging from 50 milliseconds to 450 milliseconds. As we expect, both the median latency (more generally, the 25th, 50th, 75th, and 95th percentile), as well as the deviation of the latencies increase with increasing load. Further, up to a load of 200 queries/sec the deviations of the latencies are quite small, with latencies for the 75th percentile well below 100 milliseconds.

Second, we investigate how the number of query processors on dedicated cores affects the query latency. Figure 8.7 shows four curves, each one relating to a series of experiments for a number of query processors ranging from one to four. The curves show the median latencies. The curve with the three concurrent processors
is based on the same measurements as Figure 8.6. We observe that, for all curves, the query latency increases with load. Beyond a certain load, the latencies rise steeply, which we explain with approaching the capacity of the query processing system. We also observe, for a given latency target, say 200 milliseconds, that the system capacity increases with the number of query processors. Beyond a certain number of processors, three processors in our configuration, the gain in system capacity decreases, an effect we explain by scheduling queries from an increasing number of processors towards a single database access point.

Third, we investigate the computational overhead of network search. Figure 8.8 shows the CPU utilization of the servers produced by the search nodes in function of the query load, for different number of query processors. As above, the figure shows four curves, for different numbers of query processors. The curves are based on the same measurement results as those used by Figures 8.6 and 8.7. Each curve shows a linear segment and flattens out at some query load. We attribute the sudden change of slope in a curve to the query processors becoming overloaded (which we confirmed through other measurements). Recall that we run the experiments on 24 core processors, where each processor can consume up to 4.1% of CPU capacity. The experiment shows that controlling the number of query processors is an effective way of controlling the overhead of the network search systems.

The above measurements demonstrate that our prototype can support a load of 200 queries/sec at a latency below 100 milliseconds for the 75th percentile of the queries, by using at most 5% of the CPU, when running three query processors on three dedicated cores. Based on the properties of the echo protocol, we conclude that when the system is scaled up to, say, 100'000 nodes, the above performance metrics will stay approximately the same, except for the latency, which will increase.

### 8.8 Discussion

In this paper, we have proposed a set of parameterizable matching and ranking functions for a simple query language that we developed for network search. We have shown how matching and ranking functions can be computed in a distributed and scalable way. Further, we presented the design of a network search node and reported on measurement results from a network search prototype on a cloud testbed.

The semantics for matching and ranking introduced in Section 8.4 are tailored to networked systems, but not to specific technologies. For instance, we propose specific matching rules for name resolution or ranking policies that consider freshness of objects. However, up to now, we did not consider specific matching rules for IP networks, for example, although such rules can support powerful explorative search. For instance, given an IP address as a search term, NAT-aware address matching could match the address to its translated version in other domain. Furthermore, a matching function that exploits the concept of the IP address could match an address to a subnet or vice versa. Also, a flow id could match another flow identifier that belongs to the same application. Lastly, the name of a computing device, for instance, could match data representing the DHCP server that provides the address
for the device, or it could match the AAA rules that define the security policy of the device.

The measurements from our prototype system show that it is feasible to build a network search system that can process a load of 200 global queries/sec with an overhead of less than 5% CPU load on our cloud platform. Knowing the design of the system and the properties of the echo protocol that underlies query processing, a back-of-the-envelope calculation shows that a system of 100'000 servers processing the same load would exhibit the same overhead per server and a query latency of below 1 sec.

Up to now we have developed a functionally complete, simple design of a network search system. Much work remains to be done for our design to be effective in practical scenarios. For instance, security and privacy issues need to be addressed, concepts for search space reduction need to be developed, search across multiple domains needs to be investigated, etc.
Figure 8.7: Impact of concurrency of local query processing on median latency of global queries for different query loads. The curves represent configurations for 1, 2, 3, 4 query processors, each processor execution on a dedicated core.

Figure 8.8: Computational overhead of network search: CPU usage of search node for different query loads and number of concurrent query processors.
Chapter 9

A Bottom-Up Design for Spatial Search in Large Networks and Clouds

Misbah Uddin†, Rolf Stadler†, Alexander Clemm‡
† KTH, Royal Institute of Technology, Sweden. Email: {ahmmud,stadler}@kth.se
‡ Cisco System Inc., San Jose, USA. Email: {alex}@cisco.com

Abstract

Information in networked systems often has spatial semantics: routers, sensors, or virtual machines have coordinates in a geographical or virtual space, for instance. In this paper, we propose a design for a spatial search system that processes queries against spatial information which is maintained in local databases inside a large networked system. In contrast to previous works in spatial databases and peer-to-peer designs, our design is bottom-up, which makes query routing network-aware and thus efficient, and which facilitates system bootstrapping and adaptation. Key to our design is a protocol that creates and maintains a distributed index of object locations based on information from local databases and the underlying network topology. The index builds upon minimum bounding rectangles (MBRs) to efficiently encode locations. We present a generic search protocol which is based on an echo protocol and uses the index to prune the search space and perform query routing. The response times of search queries increase with the diameter of the network, which is asymptotically optimal. We study the performance of the protocol through simulation in static and dynamic network environments, for different network topologies, and for network sizes up to 100,000 nodes. In all experiments, the overhead incurred by our protocol lies within thirty percent of a hypothetical optimal protocol. In addition, the protocol provides high accuracy under significant churn.

9.1 Introduction

In our recent work, we introduced the concept of network search, which builds upon an information-centric view of network management and is specifically suited
CHAPTER 9. A BOTTOM-UP DESIGN FOR SPATIAL SEARCH IN LARGE NETWORKS AND CLOUDS

for large-scale, dynamic environments [1, 2, 161]. A network search system allows operational and configuration data in a networked system to be searched in real-time using keywords and relational operators. We developed and implemented a peer-to-peer design of such a system, using a self-organizing network overlay without centralized components. Search queries can be initiated from any node in the overlay. Data that can be searched is organized as objects that are maintained in real-time databases inside search nodes close to the data source. Search queries are propagated across nodes to the target objects. Network search is suitable for environments where bandwidth constraints or privacy/compliance restrictions prevent the export of data to an external system.

Information associated with networked systems often has spatial semantics. This motivates us to introduce spatial concepts in network search and study spatial queries for search in networked systems. We use the Euclidean spatial model in $\mathbb{R}^n$, which defines locations as $n$-dimensional vectors, spatial entities, such as lines, hyper-rectangles, $n$-spheres, etc., and a distance function. Once we associate locations with objects, we can formulate spatial queries.

Consider a networking scenario where locations for routers, servers, and virtual machines are produced by a network coordinate system, such as Vivaldi [11]. In this case, the Euclidean distance between locations refers to the round-trip-time between the network entities at those locations. Spatial queries include finding a server that is closest to a client application or finding a server with a similar distance to a given set of clients. Second, consider a networking environment where locations refer to geographic coordinates and distances refer to geographical distances. A spatial query for this case is finding backup servers outside a given area to improve availability in case of failures. Third, consider an IoT scenario with garbage containers at different geographical locations. A spatial query is finding full containers within a certain distance from a given place, for example, to facilitate garbage collection. Fourth, consider an ICT infrastructure, whereby locations are IP addresses mapped onto $\mathbb{R}^4$ (in case of IPv4). A spatial query in this case is finding physical or virtual machines in a given address range for the purpose of security management. Lastly, consider the case where a router searches for the closest gateway in the context of performance-based routing in order to optimize service levels.

We illustrate the use of the Euclidean spatial model with two spatial queries. We assume a search space $O$ of objects, whereby each object $o \in O$ has a location $l$ in the Euclidean space $L = \mathbb{R}^n$. Two objects $o$ and $p$ have the distance $d(o,p) = \sqrt{\sum_{i=1}^{n} (o_i - p_i)^2}$. First, a range query returns all objects in $O$ whose locations intersect with a given geometric object like a point, a line, a hyper-rectangle, or an $n$-sphere. For example, the query range($l,r$) returns all objects within a distance $r \geq 0$ from location $l$. Second, a nearest-neighbor query returns $k$ objects in $O$ whose locations have the shortest distance from a given geometric object. The query kNearestNeighbors($l,k$), for example, returns $k$ objects nearest to a given location $l$. Figure 9.1 shows examples of spatial queries.

This paper presents a bottom-up design of a large-scale spatial search system that supports a range of spatial queries (like those presented above) for a Euclidean spatial model. The search is performed in a peer-to-peer fashion in a network graph of search nodes, each of which maintains a real-time database of locally sensed
9.1. INTRODUCTION

Figure 9.1: Spatial queries on a location space $L \in \mathbb{R}^2$. Locations appear as dots. The query range((5,5),2) returns all objects with locations in the disc with center (5,5) and radius 2, the point query range((5,5),0) returns objects at (5,5), and the query kNearestNeighbors((2,8),5) returns objects with locations represented by larger dots outside the disc.

objects. The design includes a protocol that creates and maintains a distributed index of object locations on search nodes using the R-tree concept [90]. It also includes a query processing protocol, which is based on an echo protocol and uses the index to perform pruning of the search space and query routing. Our design is adaptive in the sense that the distributed index is updated in response to changes to local databases and to changes in the graph of search nodes. By design, the response times of spatial queries increase with the diameter of the network, which is asymptotically optimal. Extensive simulations show the effectiveness of the index in reducing the overhead of the query processing protocol, when compared to a baseline protocol that searches all nodes for a given query. In addition, the protocol provides high accuracy under significant churn.

All spatial search systems we found in the literature follow a top-down design [86, 87, 96, 97, 100]. In these systems, the index information, which is the basis for query routing, is created top-down, by recursively partitioning the location space. The index is then used to create an overlay topology that optimizes query routing in terms of query execution time and/or query overhead on the overlay, without taking into account the underlying network. (In addition, indexes contain only pointers to information objects, and how these objects are retrieved and processed is not addressed in these works.) A top-down design, which does not take the underlying network into account, has several drawbacks. First, spatial search for applications in which search distance for query routing does not correlate with network distance for packet routing may cause more nodes to be traversed than necessary, resulting in longer query response times, increased use of network resources, and greater vulnerability to network churn, compared to a bottom-up design that is network congruent. Second, bootstrapping of the search topology requires global knowledge of the network. A bottom-up approach would allow bootstrapping through piggybacking on existing neighbor discovery mechanisms and protocols.

The spatial search system presented in this paper follows a bottom-up design. Specifically, the index information is created bottom-up from (typically a large number of) local databases on interconnected search nodes. Further, network links
between these nodes directly relate to links of the underlying routing or physical topology, enabling efficient query routing in the target environment and facilitating the bootstrapping of the search overlay. A characteristic of our design is that the objects are stored on the same node as their indexes, and object information can be retrieved locally as part of query processing. Our design is particularly suitable for environments where moving partitions of local databases is not feasible due to privacy concerns, bandwidth constraints, or other barriers.

This paper is a significant extension of results reported in [162]. In addition to more text and figures to better motivate and describe the core concepts, this paper discusses how kNearestNeighbor queries can be processed and provides performance results for such queries. Further, it extends the evaluation of our design from static to dynamic environments. Lastly, the paper includes an extended related work section.

This paper is organized as follows. Section 9.2 presents the architecture of a spatial search system. Section 9.3 describes the indexing protocol for efficient processing of spatial queries. Section 9.4 outlines the protocol for processing spatial queries. Sections 9.5 and 9.6 contain the result of the performance evaluation of our design for static and dynamic environments. Section 9.7 discusses related research and Section 9.8 gives the conclusions.

## 9.2 Architecture

![Architecture of a spatial search system](image)

Figure 9.2: The architecture of a spatial search system.

Figure 9.2 shows our architecture for a spatial search system. Its key element is the search plane, where spatial queries are processed. This plane contains a bidirectional, connected graph $G$ of search nodes. The vertices of the graph are
the search nodes and the edges are the message-passing links. Each search node maintains a local database of objects. It performs the processing of spatial queries in collaboration with other nodes. During query processing, it exchanges messages with its neighbors. The search plane forms a peer-to-peer query processing system where a query can be invoked at any node and its result is obtained from that node.

The bottom plane in Figure 9.2 represents the physical infrastructure. Each element of this infrastructure is associated with a search node, which maintains information from it in form of objects. (The figure shows a one-to-one relationship between search nodes and infrastructure nodes for clarity reasons. Our design also supports a one-to-many relationship.) The top of the figure shows the management plane, which issues the queries and processes the results.

We envision that a search node runs on the physical infrastructure, for instance on a network element, on a server, or on a network appliance. The links between search nodes are realized through link-level connections.

### 9.3 Distributed Index

In this section, we propose a distributed index structure of object locations for processing spatial queries. We also present the protocol MBRIndex, which builds and maintains this index. For better readability, we use the term “node” instead of “search node” (Figure 9.2).

The index is based on the geometric concept of the Minimum Bounding Rectangle (MBR). An MBR is a hyper-rectangle in the location space \( L \subset \mathbb{R}^n \). It is defined by two locations, \((l_1, \ldots, l_n)\) and \((h_1, \ldots, h_n)\), whereby \( l_i \leq h_i \). We aggregate a set of spatial objects through the MBR with the smallest diameter that includes all these objects (see Figure 9.3(a) and Figure (b)). We use an MBR to aggregate the locations of the objects in a node’s local database, as well as the locations of objects a node knows about. The MBR concept allows us to efficiently capture a potentially infinite number of spatial objects by giving just two locations.

Figure 9.3: (a) The MBR of a set of locations in \( \mathbb{R}^2 \) shown as dots; the locations \((l_1, l_2)\) and \((h_1, h_2)\) defines the MBR, (b) MBR\(_v\) is the minimum bounding rectangle of MBR\(_u\), MBR\(_w\), MBR\(_z\), and localMBR\(_v\), (c) The local index of node \( v \), and (d) node \( v \) and its neighbors on the spanning tree.
MBR is a common concept for indexing spatial information in a variety of areas [93, 94]. It is generally used in form of an R-tree [90]. Each node of such a tree is associated with an MBR, whereby the MBR of a non-leaf node is the minimum bounding rectangle that includes the MBRs of its children. R-trees are used in spatial databases as part of local or distributed indexes [95, 96, 97]. In contrast to all other distributed designs in the literature, each node in our design is the root of an R-tree. When describing our design, we use the term MBR-tree instead of R-tree, since the latter term is associated with a specific top-down space partitioning method.

In our design, the distributed index is built bottom-up. Each node has a local MBR, namely, the MBR of the object locations in the local database. To build an MBR-tree, we first construct a spanning tree on graph $G$ (see Section 9.3), which defines the topology of the MBR-tree. On this spanning tree, the MBR of a leaf node is the local MBR, and the MBR of a non-leaf node is the minimum bounding rectangle of the MBRs of each of its children, as well as its local MBR. As an example of a non-leaf node, MBR$_v$ in Figure 9.3(b) is the minimum bounding rectangle of MBR$_u$, MBR$_w$, and MBR$_z$, which are the MBRs of its children, and localMBR$_v$, which is its local MBR. In our design, an MBR tree forms the basis for query processing and routing for those queries that are invoked at the root node of this tree.

The MBR of a node $v$ of the MBR-tree provides information about the object locations $v$ has knowledge of. Therefore, in our design, MBR-trees form a basis for query processing and routing.

Figure 9.3(c) shows the local index $T$ of a node $v$. It is a table with two columns, where each row contains information about a node, either about $v$ or one of its neighbors on the spanning tree. The first column contains the local id of a node; the second column contains the MBR of that node. When node $v$ processes a query, it uses the MBR field of each of its neighbors on the spanning tree to decide whether or not to forward, i.e., route the query to that neighbor.

The Indexing Protocol

Algorithm 1 gives the pseudocode of the MBRIndex protocol. This protocol runs on each node and initializes and updates the MBR fields of the local index. During its execution, nodes exchange update messages with neighbors over the links of the spanning tree. The local data structures of a node $v$ are given on top (lines 1-2). The variable $N$ is initialized with the set of neighbors of $v$ on the spanning tree.

The update messages exchanged by the nodes during the protocol execution are of the form (update, $n$, MBR$_u$) (line 3), where $n$ identifies the sending node. Node $v$ starts the protocol by initializing the MBR fields of the local index and then sending an update message with its local MBR to each of its neighbors (lines 4-7). $v$ then enters a loop where it processes update messages. After reading an update message from $u$, $v$ updates the MBR field of row $u$ with the MBR contained in the message (lines 9-10). After that, for each neighbor $n$, it computes the minimum bounding rectangle of all MBRs in the local index, except the MBR of the node $n$, and sends an update with the computed MBR to node $n$ (lines 11-13).
Algorithm 1 MBRIndex. Pseudocode for node $v$. The protocol executes on a spanning tree of the network graph $G$. It initializes and updates the MBR fields of the local index of node $v$.

data structures:
1: $N :=$ set of neighbors of $v$;
2: $T :=$ local index;
message types:
3: ( update, $n$, $MBR_n$)

MBRIndex():
4: $T$.MBR$[v] :=$ local MBR;
5: for $n \in N$ do
6: $T$.MBR$[n] := (-\infty, \infty)$;
7: send ( update, $v$, $T$.MBR$[v]$) to $n$;
8: while true do
9: read ( update, $u$, $MBR_u$);
10: $T$.MBR$[u] := MBR_u$;
11: for $n \in N$ do
12: $MBRsend_n := MBR(\{ T.MBR[i] \mid i \in N \cup \{v\} - \{n\} \})$;
13: send ( update, $v$, $MBRsend_n$) to $n$;

Critical to Algorithm 1 is line 12, which can be illustrated using the example in Figure 9.3(d). The MBR that node $v$ sends to node $u$, for instance, is the minimum bounding rectangle of its local MBR, the MBR of node $z$, and the MBR of node $w$. In other words, node $v$ sends its node MBR to node $u$, assuming $u$ is its parent.

To keep the pseudocode compact, first, we omit in Algorithm 1 statements that prevent the protocol from sending unnecessary messages. For instance, before sending an update to a neighbor, a node can check whether the MBR to be sent is different from the MBR sent in the last message. In case there is no difference, the message can be omitted. Second, the algorithm does not explicitly capture changes to the local MBR. If a change in the local MBR occurs, we assume that a message ( update, $v$, newLocal$MBR_v$) is generated, which can be read by the protocol.

Constructing the Spanning Tree

MBRIndex executes on a spanning tree of the network graph $G$. There are several protocols for spanning tree construction that can be used for this purpose, for instance, the distributed Bellman-Ford algorithm [163]. To handle node churn, we need a protocol that reconstructs the spanning tree upon changes in $G$, which Bellman-Ford does not do. Such a reconstruction is possible as long as $G$ remains connected. GAP is an example of a protocol that reconstructs the spanning tree after churn events [164]. GAP is based on a self-stabilizing tree-construction protocol described in [165]. For the evaluation of our design in Section 9.5, we use an implementation of GAP to create and maintain the spanning tree.
CHAPTER 9. A BOTTOM-UP DESIGN FOR SPATIAL SEARCH IN LARGE NETWORKS AND CLOUDS

Maintaining the Local Index in Response to Node Churn Events

Algorithm 2 AdaptMBRIndex protocol. Pseudocode for node \( v \). The protocol executes on a spanning tree of the network graph \( G \). It initializes and updates the MBR fields of the local index of node \( v \).

**Data structures:**
1. \( N := \) set of neighbors of \( v \);
2. \( T := \) local index table;
3. \( R := \) vector of message count;

**Message types:**
4. (update, \( u, MBR \))
5. (join, \( u \))
6. (leave, \( u \))
7. (timer)

AdaptMBRIndex():
8. \( T.MBR[v] := \) local MBR;
9. for \( n \in N \) do
10. \( T.MBR[n] := (-\infty, \infty); \)
11. \( R[n] := b; \) ▷ \( b \) is the max. message budget per link
12. send (update, \( v, T.MBR[v] \)) to \( n \);
13. while true do
14. read message;
15. switch (message)
16. case (update, \( u, MBR \))
17. \( T.MBR[u] := MBR; \)
18. case (join, \( u \))
19. \( T.addEntry(u, (-\infty, \infty)); \)
20. \( N := N + \{n\}; \)
21. case (leave, \( u \))
22. \( T.removeEntry(u); \)
23. \( N := N - \{n\}; \)
24. case (timer)
25. for \( n \in N \) do \( R[n] := b; \)
26. for \( n \in N \) do
27. \( MBRsend_n := MBR(\{T.MBR[i]\}_{i \in N \cup \{v\} - \{n\}}); \)
28. if \( R[n] > 0 \) then
29. send (update, \( v, MBRsend_n \)) to \( n \);
30. \( R[n] := R[n] - 1; \)

Churn events we consider include a node joining the search system, a node leaving the system, and a node crashing. We outline an extension to the MBRIndex protocol, so that the extended protocol will adapt the local index of each node after a churn event. Algorithm 2 gives the pseudocode of the extended protocol.
9.4. THE QUERY PROCESSING PROTOCOL

named AdaptMBRIndex. We rely on a spanning tree protocol that reconstructs the spanning tree after such events (see Section 9.3).

Note that, we only discuss those codes in Algorithm 2 that are not presented Algorithm 1.

First, we consider a new node $u$ joining the network graph $G$. Upon joining, $u$ starts the spanning tree protocol and the AdaptMBRIndex protocol. Then, $u$ initializes the local index with a single row $(u, (-\infty, \infty))$. In the meantime, the spanning tree protocol reconstructs the spanning tree such that $u$ is included. As part of the reconstruction process, the spanning tree protocol creates a message join($u$) on each neighbor $v$ of $u$ on the spanning tree (line 5). Similarly, it creates a message join($v$) on node $u$. The message join($u$) on $v$ is processed by the AdaptMBRIndex protocol on node $v$ by adding entry $(u, (-\infty, \infty))$ to its local index (line 21). After that, $v$ sends an update message to node $u$. The message join($v$) on node $u$ is processed by the AdaptMBRIndex protocol the same way.

Second, we consider a node $u$ (with its adjacent links) leaving the network graph $G$. After this event, the spanning tree protocol reconstructs the tree. Then, it generates a message leave($u$) on all former neighbors $v$ on the spanning tree (line 6). The AdaptMBRIndex protocol processes this message by deleting the entry for row $u$ in the local index table of node $v$ (line 25). After that, node $v$ sends an update message to its neighbors on the spanning tree. In case removing $u$ from $G$ results in a partitioned graph, the AdaptMBRIndex protocol reconstructs the index for each partition of $G$.

Third, regarding the AdaptMBRIndex protocol, a node crash can be modeled as a node leaving the system. This means that, after a node $u$ crashes, its neighbor $v$ receives a message leave($u$), and the processing of this message will be the same as described above.

Furthermore, the AdaptMBRIndex protocol restricts messages rate on each link. This means that the protocol will drop all messages beyond a given maximum rate. Dropping a message leads to a (temporarily) outdated local index, which will be refreshed when the next message arrives. Limiting the message rate on a link puts an upper bound on the message overhead of the protocol. Such a policy is realistic in operational network environments, as it allows the network to scale at a sustainable protocol overhead. At the same time, this policy decreases the accuracy of query processing.

Vector $R$ has the message count per time interval for each link (line 3). When AdaptMBRIndex starts on a node, it initializes the components of vector $R$ to a value $b$, for each link to neighbor $n$ (line 11). An underlying process running on the node periodically generates a message timer. Upon receiving the message the protocol resets the components of $R$ to $b$ (lines 28-30). The protocol can send a message to a neighbor $n$, if message count $R[n]$ is greater than zero. In that case, it decreases the count by one (35-37).
Algorithm 3 SpatialSearch. The pseudocode for node \( v \). The protocol executes on a spanning tree of the network graph \( G \). It processes the spatial query \( \text{query} \), which is in abstract form. (The interface of \( \text{query} \) is presented in Algorithm 4). The query is invoked at node \( v = \text{root} \). After termination, the query result is at the root node in variable \( R \).

**data structures:**
1. \( N := \) neighbors of \( v \);
2. \( \text{parent} := \text{null} \) \( \triangleright \) parent of node \( v \)
3. \( D := \) database of local objects;
4. \( T := \) the local index;
5. \( R := \emptyset \) \( \triangleright \) (partial) result of query
6. \( \text{prunepars} := \text{null} \);

**message types:**
7. \( (\text{exp}, \text{sender}, \text{querypars}, \text{prunepars}) \);
8. \( (\text{echo}, \text{sender}, R) \);

**protocol SpatialSearch()**
9. if \( v \neq \text{root} \) then
10. \hspace{1em} receive \( (\text{exp}, n, \text{querypars}, \text{prunepars}) \);
11. \hspace{1em} \text{parent} := n, \( N := N - \{n\} \);
12. \hspace{1em} \text{query.localQuery} (\text{querypars}, D, R);
13. \hspace{1em} \text{query.computePrunePars} (R, \text{querypars}, \text{prunepars});
14. \hspace{1em} \text{query.pruneTree} (\text{querypars}, \text{prunepars}, T, N);
15. \hspace{1em} for each \( n \in N \) do
16. \hspace{2em} send \( (\text{exp}, v, \text{query}, \text{prunepars}) \) to \( n \);
17. while \( N \neq \emptyset \) do
18. \hspace{1em} receive \( (\text{echo}, n, R_n) \);
19. \hspace{1em} \text{query.updateResult} (\text{querypars}, R_n, R);
20. \hspace{1em} \text{N} := N - \{n\};
21. if \( v \neq \text{root} \) then send \( (\text{echo}, v, R) \) to \text{parent}.

9.4 The Query Processing Protocol

\( \text{SpatialSearch} \) is a protocol that processes the spatial queries discussed in Section 9.1. It is an echo protocol that executes on the spanning tree introduced in the above section. It can be started at any node, and this node becomes the root of the protocol execution tree \( S \) (which has the same topology as the spanning tree). During its execution, the protocol uses the local index to prune the execution tree. It reduces the number of nodes whose local databases are searched during the execution of a query (see Figure 9.4). When the protocol terminates, the result of the query is available at the root node.

Algorithm 3 shows the pseudocode of the SpatialSearch protocol. For better readability, we give the code for the case where the root node is identified. Further, the code includes calls to \( \text{query} \), which is an abstract object, whose interface is given
9.4. THE QUERY PROCESSING PROTOCOL

in Algorithm 4. Concrete realizations of query are shown in Algorithms 5 and 6.

The code for the SpatialSearch protocol in Algorithm 3 is given for a node \( v \).
The local data structures are shown on top (lines 1-6). \( N \) is initialized with the set of neighbors of \( v \) on \( S \); \( \text{parent} \) holds the parent of \( v \) on \( S \); \( D \) is the database of local objects; \( T \) is the local index; \( R \) holds (the partial) result of the query; \( \text{prunepars} \) contains the parameters for pruning \( S \).

During the execution of the SpatialSearch protocol, neighboring nodes exchange
two types of messages: explorer messages of the form \((\text{exp}, \text{sender}, \text{querypars}, \text{prunepars})\) and echo messages of the form \((\text{echo}, \text{sender}, R)\) (lines 7-8). \( \text{sender} \) identifies the node that sends the message; \( \text{querypars} \) holds the parameters of the spatial query; \( \text{prunepars} \) holds the parameters for pruning \( S \); \( R \) contains the partial result of the query known to \( \text{sender} \).

The code of SpatialSearch contains both the code for the root and non-root
nodes (lines 9-21). The root node starts by performing the method localQuery
on database \( D \), which computes the search result \( R \) (line 12). The protocol then computes the query-specific parameters for pruning (line 13). After that, it prunes \( S \) by removing neighbors from \( N \) that do not contribute to the search result (line 14). Then, \( v \) sends an \( \text{exp} \) message to all neighbors in the pruned set \( N \) (lines 15-16) and waits for an \( \text{echo} \) message from each neighbor in \( N \) (lines 17-18). Upon receiving such a message, it updates \( R \) by executing the method \text{updateResult} and it removes \( n \) from \( N \) (lines 18-20).

A non-root node \( v \), upon receiving an \( \text{exp} \) message sets \( \text{parent} \) to the sender
of the message and removes the sender from \( N \) (lines 9-11). Then, it performs the
operations in lines 12-20 as described above. After that, it sends an \text{echo} message
to \( \text{parent} \) with the partial result \( R \) (line 21).

(For an in-depth discussion of echo algorithms, see [166, 13]).

Algorithm 4 Abstract methods of \text{query}. \( \downarrow \) refers to an input parameter, \( \uparrow \) refers
to an output, and \( \leftrightarrow \) refers to a parameter that is both an input and an output. The
parameters have the same meaning as the parameters in Algorithm 3.

\textbf{abstract object \text{query}()} 
1: \text{query}.\text{localQuery}(\downarrow \text{querypars}, \downarrow D, \uparrow R);
2: \text{query}.\text{updateResult}(\downarrow \text{querypars}, \downarrow R_{\text{child}}, \uparrow R);
3: \text{query}.\text{pruneTree}(\downarrow \text{querypars}, \downarrow \text{prunepars}, \downarrow T, \uparrow N);
4: \text{query}.\text{computePrunePars}(\downarrow R, \downarrow \text{querypars}, \leftrightarrow \text{prunepars});

Algorithm 4 contains the interfaces of the abstract spatial query. The method \text{localQuery} executes a spatial query with parameters \text{querypars} on the local database \( D \) and returns the result in \( R \). The method \text{updateResult} merges \( R \) with \( R_{\text{child}} \) from a child node and updates \( R \). The method \text{pruneTree} removes neighbors from \( N \) using query-specific parameters and the local index \( T \). The method \text{computePrunePars} updates the prune parameters for pruning the tree \( S \).

We give two examples of spatial queries in the form of concrete implementations
of the abstract \text{query} in Algorithm 4. The first query is \text{range}(l, r), which returns
objects within distance \( r \) from a given location \( l \). Its pseudocode is given in Algo-
Algorithm 5 \textit{range}(l, r) query. \textit{range}(l, r) is a concrete implementation of the abstract \textit{query} in Algorithm 4. It returns objects within distance \(r\) from location \(l\).

\textbf{object \textit{range}}\((l, r)\):

1. \(l \in \mathbb{R}^2\) location;
2. \(r \geq 0\) distance;
3. \textbf{procedure} \textit{query}.localQuery\(([l, r], D, R)\)
4. \(R := \text{disc}(l, r) \cap D;\)
5. \textbf{procedure} \textit{query}.updateResult\(([l, r], R_{\text{child}}, R)\)
6. \(R := R \cup R_{\text{child}};\)
7. \textbf{procedure} \textit{query}.pruneTree\(([l, r], p, T, N)\)
8. \textbf{for} each \(n \in N\) \textbf{do}
9. \hspace{1em} \textbf{if} \(T.MBR[n] \cap \text{disc}(l, r) = \emptyset\) \textbf{then} \(N := N \setminus \{n\};\)
10. \textbf{procedure} \textit{query}.computePrunePars\((R, [l, r], d)\)
11. \(d := \text{null};\)

Figure 9.4: Pruning the execution tree. Query \textit{range}(l, r) is processed by node \(v\), whose set of neighbors \(N\) includes \(u, z,\) and \(w\). Algorithm 5 removes \(z\) and \(w\) from \(N\), because \(MBR_z \cap \text{disc}(l, r) = \emptyset\) and \(MBR_w \cap \text{disc}(l, r) = \emptyset\). Thus, nodes in the subtree rooted at \(z\) and \(w\) are pruned from the execution tree and the query is routed to node \(u\) only.

The second query is \textit{kNearestNeighbors}(\(l, k\)), which finds \(k\) objects closest to a given location \(l\). Its pseudocode is presented in Algorithm 6. The method localQuery initializes \(R\) with \(k\) objects in \(D\) closest to \(l\). The method updateResult merges \(R\) with \(R_{\text{child}}\), which contains the objects known to \(child\) that are within distance \(r\) from location \(l\). Note that the query \textit{range}(l, r) does not use specific pruning parameters. The method pruneTree removes neighbor \(n\) from \(N\) if the disc defined by parameters \(r\) and \(l\) does not overlap with the MBR of node \(n\) with respect to node \(v\). An example of pruning is shown in Figure 9.4.

The second query is \textit{kNearestNeighbors}(\(l, k\)), which finds \(k\) objects closest to a given location \(l\). Its pseudocode is presented in Algorithm 6. The method localQuery initializes \(R\) with \(k\) objects in \(D\) closest to \(l\). The method updateResult merges \(R\) with \(R_{\text{child}}\), which contains \(k\) objects known to \(child\) closest to the location \(l\), and then updates \(R\) with \(k\) objects in the merged set closest to \(l\). The method pruneTree prunes the set \(N\) if the prune parameter \(R_{\text{prune}}\) has \(k\) objects. It removes neighbor \(n\) from \(N\), if distance between location \(l\) and the MBR of node
9.5 PERFORMANCE EVALUATION OF SPATIALSEARCH IN A STATIC ENVIRONMENT

Algorithm 6 kNearestNeighbors(l,k) query. kNearestNeighbors(l,k) is a concrete implementation of the abstract object query in Algorithm 4. It returns k objects closest to location l.

object kNearestNeighbor(l,k):
1: \( l \in \mathbb{R}^2 \) location;
2: \( k \in \mathbb{N} \);
3: procedure query.localQuery([l,k],D,R);
4: \( R \leftarrow k \) objects in D closest to \( l \);
5: procedure query.updateResult([l,k],R_{child},R);
6: \( R \leftarrow k \) objects in \( R \cup R_{child} \) closest to \( l \);
7: procedure query.pruneTree([l,k],R_{prune},T,N);
8: if \( |R_{prune}| = k \) then
9: for each \( n \in N \) do
10: if dist(T.MBR[n],l) \( \geq \) dist(l_o,l) then \( \forall o \in R_{prune} \)
11: \( N \leftarrow N \setminus \{n\} \);
12: procedure query.computePrunePars(R,[l,k],R_{prune})
13: \( R_{prune} \leftarrow k \) objects in \( R \cup R_{prune} \) closest to \( l \);

n is larger than the distance between \( l \) and the location of every object in \( R \).

9.5 Performance Evaluation of SpatialSearch in a Static Environment

We have evaluated the performance of the SpatialSearch protocol using Peersim, a Java-based platform for simulating large-scale peer-to-peer systems [167]. We model a network search system as a (network) graph of search nodes, each having a local database with objects. The SpatialSearch protocol executes on these search nodes and communicates via sending messages over the links of the network graph.

To perform the evaluation, we implemented a spatial search system based upon (a) a coordinate generator that produces the locations for nodes of a network graph; (b) a topology generator that generates network graphs for different network sizes and connectivity policies; (c) a GAP protocol that creates a spanning tree on the network graph [164]; (d) the MBRIndex protocol, which creates and maintains the distributed location index; and (e) the SpatialSearch protocol for point, range, and kNearestNeighbor queries.

Performance Metrics: We evaluate the performance of the SpatialSearch protocol for a specific query by counting the number of nodes whose local databases are searched during the execution of the query, and we compare the result with two baselines. The first baseline is the size of the network graph, which is an upper bound for the performance of SpatialSearch and which is the performance of the protocol that searches all local databases. We call this protocol CompleteSearch. It can be implemented as a simple echo algorithm. The second baseline is the minimal
CHAPTER 9. A BOTTOM-UP DESIGN FOR SPATIAL SEARCH IN LARGE NETWORKS AND CLOUDS

number of nodes whose databases must be searched to process a search query. We call such a hypothetical protocol $OptimalSearch$.

In the evaluation, we compute the efficiency metric

$$E = \frac{N_{SpatialSearch} - N_{Optimal}}{N - N_{Optimal}}$$

whereby $N_{SpatialSearch}$ is the number of nodes whose local databases are searched by SpatialSearch when processing the query, $N$ is the size of the network graph, i.e., the number of nodes, and $N_{Optimal}$ is the minimal number of nodes to be searched. $E = 0$ is the efficiency of $OptimalSearch$ and $E = 1$ that for $CompleteSearch$. The metric $E$ allows us to compare the performance of SpatialSearch for different network sizes, and it indicates how well the protocol performs compared to $CompleteSearch$ (upper bound) and $OptimalSearch$ (lower bound). For instance, a value of $E = 0.2$ means that the protocol has an overhead of twenty percent (of the difference between the baselines) relative to $OptimalSearch$ and saves eighty percent (of this difference) relative to $CompleteSearch$. Further, the metric $E$ not only implies efficiency, but also scalability. A small value of $E$ means that the system has a greater capacity to support a larger number of concurrent spatial queries.

**Network Graph Topology:** For the evaluation, we produce network graphs with 512, 2048, 8192, 32'536, and 131'072 nodes. For each node, we generate a location in $\mathbb{R}^2$ uniformly at random within a square of size $s=16$ for a network with 512 nodes, $s=32$ for a network with 2048 nodes, $S=64$ for a network with 8192 nodes, $s=128$ for a network with 32'536 nodes, and $s=256$ for a network with 131'072 nodes. Note that the average node density is the same for all network sizes.

We use two different graph topologies in the evaluation. First, the Barabási-Albert topology, which is a scale-free topology that follows the degree distribution $p(k) \sim k^{-3}$, $k = 1, 2, 3, \ldots$ [168]. Second, the nearest-neighbor topology, which is constructed by connecting each node of the graph to its $k$ nearest neighbors. For the experiments, we set $k = 20$, which produces connected graphs with a high probability. Before the experiments, the graphs are tested for connectivity.

The Barabási-Albert topology is oblivious to node locations, while the nearest-neighbor topology is constructed based on location awareness. We expect the performance of the protocol on the nearest-neighbor topology to be better than on the Barabási-Albert topology, since the area of the MBR of a subtree of the spanning tree is in expectation smaller for a nearest-neighbor topology than for a Barabási-Albert topology. Comparing the performance of the protocol for both topologies with the same number of nodes is fair in the sense that the spanning trees of these topologies have the same number of nodes and edges.

**Local MBRs:** We assume that each search node is aware of its location and maintains objects in databases that are nearby. For simplicity, we assume for the evaluation that the location of all objects in the local database is the same as the location of the node.

**SpatialSearch for range($l,r$) queries:** Figure 9.5 shows the measurement results of the SpatialSearch protocol for range($l,r$) queries executing on both topologies and different network sizes. The top two plots in Figure 9.5 show measurements on the Barabási-Albert topology, the bottom two on the nearest-neighbor topology.
9.5. PERFORMANCE EVALUATION OF SPATIALSEARCH IN A STATIC ENVIRONMENT

Figure 9.5: The efficiency metric $E$ of the SpatialSearch protocol for range($l,r$) queries with location $l$ and distance $r$. The top two plots show measurements on the Barabási-Albert topology, the bottom two on the nearest-neighbor topology. The plots on the left show efficiency vs distance, for different network sizes. $E = 0$ is the efficiency of OptimalSearch and $E = 1$ is that of CompleteSearch. The plots on the right show number of search nodes vs distance, for networks of size 131'072.

Each measurement point on a curve gives the average efficiency of 1000 queries, whereby each query is invoked with a start node and a location $l$, both of which are chosen uniformly at random on the network graph. The confidence intervals of the measurement points are too small to be shown in the figures. To create the curves in Figure 9.5, we produce measurement points for $r = 0, 1, 2, \cdots, 350$.

A query with $r = 0$ corresponds to a search for objects with a given location $l$. Figure 9.5(a) shows that the efficiency of this query is about 0.12 for the Barabási-Albert topology for all tested network sizes. With increasing $r$, the efficiency increases to about 0.31 and then decreases to approach 0. For $r \geq \sqrt{2}s$ ($s$ is the side of the square area, see above), the efficiency of the protocol becomes undefined and the performance of OptimalSearch, SpatialSearch, and CompleteSearch becomes the same. This can be seen in Figure 9.5(b), which shows the performance of SpatialSearch, measured in the number of locally searched databases, for a network of 131’072 nodes. The performance of the protocol on the nearest-neighbor
CHAPTER 9. A BOTTOM-UP DESIGN FOR SPATIAL SEARCH IN LARGE NETWORKS AND CLOUDS

topology, presented in Figure 9.5(c) and Figure 9.5(d), is closer to OptimalSearch than on the Barabási-Albert topology. Otherwise, the protocol shows the same qualitative behavior as when it executes on the Barabási-Albert topology.

**SpatialSearch for kNearestNeighbors(l, k) queries:** Figure 9.6 shows the measurement results of SpatialSearch for kNearestNeighbors(l,k) queries executing on both topologies and different network sizes. The top two plots in Figure 9.6 show measurements on the Barabási-Albert topology, the bottom two on the nearest-neighbor topology. The plots on the left in Figure 9.6 contain efficiency vs k curves, for different network sizes. $E = 0$ is the efficiency of OptimalSearch and $E = 1$ is that of CompleteSearch. The plots on the right in Figure 9.6 show number of search nodes vs k curves, for networks of size 131’072.

Each measurement point on a curve gives the average efficiency of 1000 queries. Each query is invoked with a start node and a location l, which are chosen uniformly at random on the network graph. The confidence intervals of the measurement points are not shown in the figures, since they are too small.

A query with $k = 1$ corresponds to a search for the object closest to a given location l. Figure 9.6(a) shows the efficiency of this query is between 0.19 and 0.26 for the Barabási-Albert topology and different network sizes. For $k > 1$, the efficiency monotonically increases with increasing k. Figure 9.6(b) shows the performance of SpatialSearch for this query, measured in the number of locally searched databases, for a network of 131’072 nodes. The performance of the protocol on the nearest-neighbor topology, presented in Figure 9.6(c) and Figure 9.6(d), is slightly better than on the Barabási-Albert topology. Otherwise, the protocol shows the same qualitative behavior as when it executes on the Barabási-Albert topology.

**Results:** The described experiments allow us to draw the following conclusions. First, for range queries with small values of r and kNearestNeighbors queries with small values of k, SpatialSearch is clearly more efficient than CompleteSearch. In the case of large values of r and large values of k, SpatialSearch performs close to CompleteSearch, which means that almost all network nodes need to be searched. Second, for the same value r, SpatialSearch performs better on a larger network than on a smaller network. The same is true for the value k in kNearestNeighbors query. Third, SpatialSearch performs significantly better on the nearest-neighbor topology than on the Barabási-Albert topology in terms of the number of nodes whose databases are searched. This is an artifact of the topology design, as explained above. For range queries, SpatialSearch performs close to OptimalSearch.

Comparing the graphs in Figure 9.5 to those in Figure 9.6 shows a striking difference in the shape of the curves. This can be explained by the fact that the OptimalSearch protocol for kNearestNeighbor queries searches increases with k nodes, while for range queries, it increases with $r^2$.

In a number of experiments that are not reported in detail here, we measured the heights of the execution tree of search queries, which correlate with the execution times of the queries. The height of an execution tree is bounded by the diameter of the network graph, and, therefore, the response times of search queries depend on the topology of the network graph. We found that, for both range and kNearestNeighbors queries, the response times approach those of CompleteSearch, which can still be quite short under realistic assumptions (for instance, for a system size
9.6. PERFORMANCE EVALUATION OF SPATIALSEARCH IN A DYNAMIC ENVIRONMENT

Figure 9.6: The efficiency of SpatialSearch for kNearestNeighbors(l,k) queries with location l and parameter k. The top two plots show measurements on the Barabási-Albert topology, the bottom two on the nearest-neighbor topology. The plots on the left show efficiency vs parameter k, for different network sizes. E = 0 is the efficiency of OptimalSearch and E = 1 is that of CompleteSearch. The plots on the right show number of search nodes vs parameter k, for networks of size 131,072.

of 8192 nodes, the average height of the execution tree is around 10 hops). Second, we found that the response times on the Barabási-Albert topology are significantly shorter than those on the nearest-neighbor topology, which can be explained by the fact that the diameter of a Barabási-Albert topology increases with \( O(\log n) \), whereas the diameter of a nearest-neighbor topology increases with \( O(n^{1/2}) \).

9.6 Performance Evaluation of SpatialSearch in a Dynamic Environment

In this section, we investigate the accuracy of SpatialSearch queries in dynamic environments. We model events in a dynamic environment as nodes joining or leaving the search system, objects changing their locations, or objects being added to or removed from local databases. The design of the spatial search system presented in the previous sections is adaptive to such events in the sense that the system converges to a state where the distributed index reflects the changes caused by the events. This is true even when further events occur during the time when the system adapts to changes. When a node joins or leaves the system, the MBRIndex protocol updates the distributed index accordingly. The same applies whenever the local MBR of a node changes, which can occur when a locally stored object changes its location, a new object is added to the local database, or removed from it.
CHAPTER 9. A BOTTOM-UP DESIGN FOR SPATIAL SEARCH IN LARGE NETWORKS AND CLOUDS

For this part of the investigation, we use the AdaptMBRIndex protocol, which restricts the maximum message rate per link to $R_u$ (Algorithm 2).

The simulated system in this study includes the AdaptMBRIndex protocol and the SpatialSearch protocol. These protocols executes in the simulation environment and react to external churn events. We measure the accuracy of queries that execute in this environment.

**Performance Metric:** We define the accuracy of a query executed in a dynamic environment using the recall metric from information retrieval [169]:

$$\text{accuracy} := \frac{O_{\text{SpatialSearch}}}{O_{\text{correct}}}.$$  

Here, $O_{\text{SpatialSearch}}$ refers to the number of objects returned by the protocol that satisfy the query, and $O_{\text{correct}}$ is the total number of objects in the system that satisfy the query.

In this set of experiments, we do not investigate node churn, but focus on changes associated with objects changing their locations or objects being added to or removed from the local databases. Such events result in changes to local MBRs.

**Simulation Model:** The simulation processes two types of external events. First, it processes churn events, whereby each such event represents a change to a local MBR. We model the effect of a change to the local MBR $(l, h)$ as follows. With equal probability, $l$ or $h$ is chosen; in case of choosing $l$, the value changes to $l + \text{random}(-\delta, \delta)(h - l)$ for a given $\delta$; in case of choosing $h$, an analogous change is made. For our experiments, we chose $\delta = 0.5$. Second, the simulation processes query invocation events for range queries with parameters $l$ and $r$. Each query is invoked on a node selected uniformly at random.

Messages of the AdaptMBRIndex protocol that traverse a link between two nodes incur a uniform, nonzero communication delay, which determines the evolution of the distributed index over time. The messages of the SpatialSearch protocol incur no communication delay, which means that messages are exchanged between nodes “instantaneously”. A query, therefore, is executed on a snapshot of the distributed index. This simplified simulation model is based on the assumption that the execution time of a query is small compared to the convergence time of the global index after a change event. This simplification significantly shortens the execution time of the simulation and allows us to simulate very large networks on a single high performance server. For the same reason, we assume that the processing delay of a protocol message is negligible.

We compute the accuracy of a range query as follows. $O_{\text{SpatialSearch}}$ is computed during the protocol execution, by intersecting the area of the local MBR with that of the query range, for all nodes whose local databases are searched, and summing up these intersections. $O_{\text{correct}}$ is computed by applying the same method to all nodes in the system.

**SpatialSearch for range$(l, r)$ queries under churn**

For the evaluation, we produce network graphs with 512, 2048, 8192, 32768, and 131’072 nodes. The location of each node is generated uniformly at random within a square of size $s=16$ for a network with 512 nodes, $s=32$ for a network with 2048 nodes, $s=64$ for a network with 8192 nodes, $s=128$ for a network with 32768 nodes, and $s=256$ for a network with 131’072 nodes, which ensures a uniform node density across all network sizes. Network graphs are produced following the Barabási-Albert topology described earlier.
9.6. PERFORMANCE EVALUATION OF SPATIALSEARCH IN A DYNAMIC ENVIRONMENT

Figure 9.7: Simulation results of the SpatialSearch protocol for different ratios $R_c/R_u$ and different network sizes. Figure 9.7(a) shows the results for $R_c/R_u$ in the interval $[0,40]$; Figure 9.7(b) displays the results for the partial interval $[0,0.5]$.

For each simulation run, a range$(l,r)$ query is selected and executed. Each query is invoked with a start node and a location $l$, both of which are chosen uniformly at random on the network graph. The parameter $r$ is set to 4 for all experiments.

We measure the accuracy of the SpatialSearch protocol for different values of the ratio $R_c/R_u$. $R_c$ represents the rate of churn events per node. As explained above, a churn event results in a change to the local MBR, caused either by a change of an object location or the addition of an object to (or the removal from) the local database. $R_u$ stands for the maximum message rate per link (see above). We can interpret $R_c$ as the rate of disruption to the distributed index, and $R_u$ as the rate of repair of this index. Since our simulation assumes instantaneous processing of messages, the accuracy of a query for given values of $R_c$ and $R_u$ is the same as for the values of $a \cdot R_c$ and $a \cdot R_u$ for $a > 0$. Therefore, it is sufficient to measure the accuracy of a query in function of the ratio $R_c/R_u$.

Figure 9.7 shows the simulation results of the SpatialSearch protocol for different ratios $R_c/R_u$ and different network sizes. Figure 9.7(a) shows the results for $R_c/R_u$ in the interval $[0,40]$, while Figure 9.7(b) displays the results for the partial interval $[0,0.5]$. Each measurement point gives the average accuracy of 1000 query runs. The confidence intervals of the measurement points are too small to be shown in the figures.

We observe from the figures that the accuracy decreases monotonically from one towards (close to) zero as the ratio $R_c/R_u$ increases from zero towards large numbers. As expected, if there is no churn in the network, i.e., $R_c = 0$, then the accuracy is one. Similarly, as the churn increases to larger values, the accuracy falls to a small value. (It cannot be zero, because of the nonzero probability that the start node and the parameter $l$ are the same.) Further, we observe that SpatialSearch has a higher accuracy on a smaller network than on a larger network.

Figure 9.7(b) shows the accuracy of the protocol for a possible operational range of a network search system. For instance, we can see that, for a network size of 131'072 nodes, the accuracy stays above 0.9 for $R_c/R_u$ below 0.1. Assuming $R_u$ is one message per second, the protocol returns 9 of 10 correct result objects on average, while the network experiences up to 1 churn events per nodes every 10 seconds. We draw the conclusion that the system for the chosen configuration
provides surprisingly accurate results under significant churn.

There is a tradeoff between the accuracy of the search results and the rate of repair of the local index, which is controlled by the maximum message rate per link. The results presented in Figure 9.7 confirm our intuition that the higher the rate of repair, the better the accuracy of the search results and vice versa.

How can we characterize the objects that are missed when the protocol executes a range query under churn? Our intuition says that the longer the distance between the result object and the location parameter \( l \), the larger the probability that the protocol misses the object due to temporary inconsistency of the index. The simulation results given in Figure 9.8 are in conformance with our expectation. For instance, when running a query \( \text{range}(l, r = 4) \) on a network with 131,072 nodes for \( R_c/R_u=0.1 \), an object at a distance of 1 or larger has a probability close to 0 or larger to be missed; if the object has a distance of 3.8 or larger, the figure increases to 0.9 percent.

![Figure 9.8](image)

Figure 9.8: The cumulative distribution of the distance between a missed object and the location parameter \( l \) of the query. Here, \( r=4 \) and \( R_c/R_u=0.1 \).

9.7 Related Work

Euclidean spatial models have been used in different contexts, including geographic information systems [8], sensor networks [9], and network-aware overlays [12]. The first two types of systems use geographic coordinates. The last one uses so called network coordinates, which are virtual. Examples of network coordinate systems are Vivaldi [11], Meridian [85], and Pharos [10]. In these systems, the Euclidean distance between the locations of two nodes represents the end-to-end delay between them.

Spatial queries that are used with the Euclidean spatial models include the point query, which returns objects at a given location, the range query, which returns all objects in a given spatial object, the nearest-neighbor query, which returns \( k \) objects nearest to a given location, and the centroid query, which returns the objects at the centroid of a given set of locations [85, 12].
A number of peer-to-peer frameworks have been developed that maintain a distributed index for processing spatial queries. Some of these frameworks use Distributed Hash Tables (DHTs) for query routing [86, 87, 88]. Scrap and PRoBe are two well-known examples of systems whose indexes are based on DHTs [86, 87]. They use the join and the leave operations of the DHT to construct and maintain the index in a decentralized way. In the case of Scrap, the multi-dimensional location space is mapped onto a single dimension in the form of a space-filling curve, i.e., the z-curve, which defines a single-dimensional identifier space that is managed by a DHT framework called skip graph [89]. The problem with this approach is that Scrap does not guarantee the preservation of locality in higher-dimensional location spaces. (The preservation of locality means that two locations that are close to each other either share the same responsible peer, or their responsible peers are neighbors.) For a range query, for instance, the identifiers that satisfy the query may not be found on a contiguous subgraph of peer nodes, and, therefore, the query overhead may be large. In our design, locality is always preserved, and query processing is thus limited to a, oftentimes small, contiguous subgraph of peer nodes.

Second, in the case of PRoBe, the multi-dimensional location space is directly mapped on to an identifier space of the same dimensionality, which is managed by the Content Addressable Network (CAN) [76]. CAN is a peer-to-peer framework whose peers are organized into a \( k \)-dimensional torus. Each peer is responsible for all locations within a hyper-rectangle and maintains a routing table of peers responsible for adjacent hyper-rectangles. All hyper-rectangles in CAN form a partitioning of the location space. While the PRoBe design preserves locality, query processing can still result in high overhead and high latency, for instance, when a peer invokes a point query for a location that is far away from its hyper-rectangle. In fact, due to the underlying CAN design, the latency for a point or a range query in PRoBE increases with \( O(k \sqrt{n}) \) for a network of \( n \) nodes. In contrast, the latency of such a query in our design increases with \( O(\log n) \) (given that the diameter of the network graph grows with \( O(\log n) \)), which is significantly smaller. This means that our design enables much faster query processing in large networks.

Index structures have been developed to efficiently process spatial queries, first for centralised and later for distributed systems. The most widely used index structure is the R-tree [90, 91, 92]. During the last decade, designs have been devised to use R-trees in peer-to-peer frameworks that underlie large-scale spatial information systems [96, 97, 98]. Two well-known examples of peer-to-peer frameworks that are based on R-trees are VBI-tree and DP-tree [96, 97]. The first framework, VBI-tree, maps the R-tree onto a tree of peer nodes of the same height. A node \( v \) of the peer tree stores the MBRs of some nodes of the same level on the R-tree, as well as the local databases of some leaf nodes of the R-tree. In addition, \( v \) stores the MBRs and the network addresses of all nodes on the path from \( v \) to the root of the peer tree. In the second framework, DP-Tree, a peer node stores all MBRs of the path from a leaf node of the R-tree to the root. The above two works contain also distributed algorithms to process spatial queries using these R-tree based indexes. In both of the above frameworks, the index is built starting from a given R-tree, which is then distributed to peer nodes, either by a human administrator or by using a centralized algorithm. In our design, in contrast, the index is constructed bottom-up, using two
CHAPTER 9. A BOTTOM-UP DESIGN FOR SPATIAL SEARCH IN LARGE NETWORKS AND CLOUDS

protocols. First, a spanning tree is built on the network of peers and, second, the distributed index is constructed through information exchange along the links of this tree.

Recently, a comprehensive peer-to-peer framework for processing spatial queries called MIDAS has been proposed [100]. MIDAS uses an index called k-d tree [101]. Both a k-d tree and an R-tree partition the space into axis-aligned regions. The key differences are as follows. A k-d tree describes a complete partitioning of the location space, while an R-tree generally does not cover the whole space. The MBRs of an R-tree can overlap, while the bounding rectangles of a k-d tree do not. The main advantage of a k-d tree index is that the addition or removal of objects does not require a change in index. Such a change is only needed when a new peer is added or removed. In contrast, in systems that use an R-tree index, the addition or removal of objects may require a change in the index. The main advantage of the R-tree over the k-d tree is that it supports extended objects, such as line segments, rectangles, etc., while the k-d tree only supports point objects (without substantial modification). In addition, query processing in a system using an R-tree can be more efficient in terms of overhead and execution time, if the object locations are clustered.

In the MIDAS design, the leaves of the k-d tree define the partitions of the location space, each of which is allocated to a peer. For query routing, each peer \( u \) has the knowledge of another peer \( v \) in each subtree for which there exists no larger subtree that contains \( v \) but not \( u \). The MIDAS design contains algorithms for point queries, range queries, and nearest-neighbor queries. Query latency increases with \( O(\log n) \) for a network of \( n \) peers.

While MIDAS has a comprehensive and an elegant design, it is not suitable for our purpose, since the index structure in our case requires bottom-up construction (starting from a network of distributed databases whose MBRs may overlap). For this reason, our design requires an index that is based on an R-tree.

We point out that all peer-to-peer frameworks for processing spatial queries in the current literature address the problem of efficiently processing queries on a very large spatial database. These frameworks solve the problem by mapping the database onto a set of nodes and creating an overlay topology optimised for query routing. In contrast, our spatial search system solves the problem of efficient query processing on a given network of local spatial databases. Our approach is to build a distributed index structure bottom-up and use the given network topology for query routing.

9.8 Conclusion

We have proposed a peer-to-peer design for a spatial search system. In contrast to recent works, our design is bottom-up, which makes query routing network aware. We present an indexing protocol that creates and maintains a distributed index of object locations within a network of search nodes. The protocol adapts to object and node churn. The index is built using the MBR concept, which allows to efficiently encode, process, and transport spatial information. The SpatialSearch protocol
prunes the search space and performs query routing. It supports a range of spatial queries, such as point, range, and k-nearest neighbor queries, which are implemented by realizing an abstract search object.

We have studied the performance of the SpatialSearch protocol through simulation with respect to three metrics, namely, (a) efficiency in terms of the number of search nodes involved in processing a query, (b) the query response time, and (c) the accuracy of query results under churn. Regarding efficiency, we found that the overhead of the protocol is close (within thirty percent) to that of a hypothetical optimal protocol, for all investigated scenarios; the overhead barely changes when the network size increases from 512 to over 100,000 nodes; the protocol is specifically efficient when the neighboring nodes on the network graph are close in the location space. With respect to the query response time, the protocol is asymptotically optimal by design. We found also that the response times for our protocol are very close to those of an echo-based protocol that searches all nodes. Finally, regarding the accuracy of query results, our measurements show that our protocol remains quite accurate under churn. For instance, for a churn rate of one local MBR change per node every 10 seconds, the accuracy is above 90 percent for range queries. The accuracy does not change significantly with increasing network size. The probability that an object is missing from the result of a range query increases with the churn rate and the distance of the object from the center of the query disc. Overall, we found that our protocol scales well to network sizes above 100,000 nodes with respect to the three investigated performance metrics.

There are a number of ways in which the design presented in this paper can be extended. One aspect concerns mobility. Our design is geared towards scenarios in which object locations change slowly. The current design is not suited for domains where nodes or objects are mobile, for example, when nodes are mounted inside a vehicle. In such a case, neighbor relations or locations are constantly changing, requiring continuous adjustments to the MBR-tree. Optimizing this recomputation and allowing for continuous incremental adaptation of the MBR-tree to changing locations is an area for future work.

We also plan to integrate spatial search into the framework of network search, which means extending the query language of network search, including a matching and ranking semantics for spatial queries, integrating the respective architectural designs, and producing an integrated prototype.

One aspect is to extend the SpatialSearch protocol with functionality for ranking and scoring of search results. Many applications of spatial search may not be concerned with obtaining a comprehensive set of results, but with the k most relevant results, whereby relevance is determined by factors like spatial relations among objects, freshness of information, joint attributes among objects, etc. An elegant to implement ranking is to develop a concrete realization of the abstract query object. Using this approach, the complexity of the ranking functionality can be localized in the concrete query and the SpatialSearch protocol does not need to be changed.

Another aspect worth investigating concerns the scalability of our design with respect to its efficiency in use of “search spectrum.” As pointed out earlier, the bottom-up approach to our design leads to network congruency, which, among other things, facilitates concurrent searches that utilize the same “search spectrum” in
different areas of a networked system. One aspect to investigate concerns the relationship between the distribution of searches, search diameters, and the number of searches that can be utilized overall, analogous to assessments of communication in cellular networks involving dimensioning of cells, distribution of calls, and utilization of frequency spectrum.

Finally, we see interesting use cases when applying our design and concepts to the IoT domain.

Appendix: Properties of the MBRIndex, AdaptMBRIndex, and SpatialSearch protocols

We discuss the key properties of the protocols introduced in this paper and outline the proofs of these properties. These protocols are built upon basic distributed algorithms (namely, Echo, tree-based aggregation, and GAP), and their properties rely on the properties of those. Throughout this appendix, we use the term protocol with the same meaning as distributed algorithm (as is often done in the literature).

We assume that the algorithms execute on a connected network graph with bidirectional links. Each node on this graph has a global identifier, and it can distinguish its neighbors on the graph. Neighboring nodes can exchange messages, which are read in the order they are received. We assume an asynchronous execution model with bounded delays. The reader may consult a textbook on distributed algorithms for more details on execution models [163, 166]. We further assume a set of underlying services for detecting nodes joining, leaving, or failing, local MBR update, message delivery, and timeout. These services deliver the output to the local process in queue as messages of the form (tag, \(Arg_1, \ldots, Arg_m\)).

The two version of the indexing protocol, MBRIndex and AdaptMBRIndex, are modifications to and extensions of the GAP protocol [164]. GAP is a tree-based aggregation protocol that constructs a BFS spanning tree on a connected graph \(G = (V, E)\) and dynamically computes a global function \(F(x_1, \ldots, x_n)\) of local values \(x_i, i = 1, \ldots, n = |V|\). The values \(x_i\) can change at discrete points in time. The result of the global function \(F\) is available and updated at the root node of the spanning tree. GAP is self-stabilizing in the sense that, from any initial state, when no further changes in local variables occur, the system will converge in finite time to a state where the protocol has constructed a BFS spanning tree on \(G\), and the result of \(F(x_1, \ldots, x_n)\) is available at the root node. GAP has a control parameter \(R\), which controls to the maximum message rate over the link between two neighboring nodes.

The MBRIndex protocol restricts the GAP protocol in the sense that (a) it assumes that \(G\) is a tree, and, therefore, does not construct a spanning tree (GAP assumes that \(G\) is a connected graph); (b) it does not consider node churn (GAP has functions for node joining or leaving); and (c) it does not consider rate limitations (GAP allows to control the maximum message rate). At the same time, MBRIndex is an extension of GAP in the sense that every node on the spanning tree \(G\) is the root of an aggregation tree with the same topology as (the spanning tree) \(G\). Similarly, the AdaptMBRIndex protocol restricts as well as the extends the GAP
protocol, and it uses the same mechanisms as GAP to achieve robustness to churn and to control the maximum message rate.

Based on their similarities to GAP, both MBRIndex and AdaptMBRIndex have the following properties.

1. Starting the protocol with the local indexes in arbitrary state, when no further changes in local MBR occur, the local indexes will converge in finite time.

2. Once the local indexes have converged, for any node $v$ of (the tree) $G$, $v$ is the root of an aggregation tree. The global index, namely, the value of MBR(local MBR$_1$, ···, local MBR$_n$), can be computed from $v$’s local index.

The SpatialSearch protocol is a version of the Echo protocol [132, 13]. An Echo protocol that executes on a connected graph $G$ has the following properties.

1. The protocol creates a spanning tree with the invoking node as the root.

2. The protocol terminates in at most $2 \times D$ rounds, while $D$ is the diameter of $G$.

Specifically, SpatialSearch is an Echo protocol that performs tree-based aggregation of local variables. This means that it computes a global aggregation function $F(x_1, \cdots, x_n)$ at the root node.

Since SpatialSearch is an Echo protocol, it has properties 1 and 2. The function $F$ computes the result of a spatial query through tree-based aggregation of local variables. The local variable $x_v$ is the result of executing the query against the local database of node $v$. The aggregation function is dependent on the specific spatial query. In the case of the range query, the aggregation function is the union of the result sets.

SpatialSearch prunes the execution tree of the Echo protocol to reduce the protocol overhead. Based on the properties of the local index, the protocol prunes only nodes that do not contribute to the query results. As a consequence, the protocol computes the correct result of the query. This statement is true under the assumption that all local indexes have converged.
Bibliography


[62] N. Leavitt. “Will NoSQL databases live up to their promise?”.


“On the elasticity of NoSQL databases over cloud management platforms”.

[65] W. Vogels. “Eventually consistent”.

“Dynamo: Amazon’s highly available key-value store”.

“Bigtable: A distributed storage system for structured data”.

“Cassandra: A structured storage system on a p2p network”.

“Serving large-scale batch computed data with project Voldemort”.

“Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web”.

“Electoral voting protocol-a quorum-based approach for replica control”.

BIBLIOGRAPHY


“libvirt 0.7.5 - Application Development Guide”. http://libvirt.org/guide/. [visited February 1, 2016].


