Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2011-030 June 7, 2011

Scalable Information-Sharing Network Management

Nina X. Guo

Scalable Information-Sharing Network Management

by

Nina X. Guo

S.B., C.S. M.I.T., 2010

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

June 2011

Author: _____

Department of Electrical Engineering and Computer Science
May 20, 2011

Certified by: _____

Dr. Karen Sollins
Principal Research Scientist
Thesis Supervisor
May 20, 2011

Accepted by: _____

Dr. Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

Scalable Information-Sharing Network Management

by

Nina X. Guo

Submitted to the
Department of Electrical Engineering and Computer Science

May 20, 2011

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

This thesis analyzes scalable information-sharing network management. It looks into one of the large problems in network management today: finding information across different network domains. Information-sharing network management is a method to solving the problem, though it is important to make it scalable. The solution proposed uses the Publish-Subscribe Internet Routing Paradigm (PSIRP) inter-domain design as the base structure. The design borrows from Border Gateway Protocol ideas and uses the Chord protocol as one of the key methods of finding information. The conclusion after analyzing the scalability of PSIRP is that its use of Chord gives it an advantage that allows a $O(\log^2 N)$ tradeoff between performance and distribution.

Thesis Supervisor: Dr. Karen Sollins

Title: Principal Research Scientist

Acknowledgements

I would like to thank my advisor, Karen, for all the guidance she has given me. She was a constant source of encouragement during my MEng year and was always patiently there to lend me her support and understanding. She opened my eyes to research, graduate school, and the world. I learned many valuable life lessons from her. More than an advisor, she has become a friend I can talk to. I am very grateful to have been her advisee.

I would also like to thank my mom and dad for their endless love and support. Without them I would not be where I am today.

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Problem in Network Management

With the evolution and growth of the internet, the world is becoming more and more connected. This is made possible by networks. There are many different kinds of networks that serve different purposes ranging from personal home usage to company usage to public usage. It is no surprise, then, that numerous ways of managing these networks have been developed. Much of these methods, however, only correspond to local domains. Three problems arise from the current setup of network domains when we start traversing different domains to find information:

1. The local domains are isolated and all work independently from one another. Different enterprises have their own security and policies, which keeps data hidden from each other. Thus, interacting between local domains is difficult because they were not built to interact with each other.
2. There is a wide distribution of independent domains. Just by looking at the top level, every company/institute/household most likely has its own network domain. Because of the numerous different local networks, when a client connects to a remote domain it's very likely that the path to the remote domain connects to several other domains. Finding something amongst all these different domains poses a challenge.
3. The Internet is the composition of many of those otherwise independent local networks. While the domains work fine by themselves, the problem comes when people do things over the Internet, which spans a number of these otherwise independent networks.

In order to see the importance of the two aforementioned problems, consider a scenario where we want to be able to find something in another domain. Suppose Bob were an employee at Company A but needed to work remotely at a hotel. He needs to find a way to access the company's Internet service provider (ISP). The hotel network most likely does not have a direct

connection to Company A's ISP. This means he would need to first connect to the hotel's ISP. The hotel's ISP then sends packets to some backbone ISP, which finally sends them to Company A's ISP. Also, suppose that his connection suddenly failed and that he needed to know why it went down. Normally, if Bob were working locally, he would have been able to contact his company's network manager and ask her for consultation. However, because he is in a different network, there is no sure way to find consultation. He can contact either the hotel's ISP or the company's ISP if the failure happened on either side. However, if the failure happened at an ISP in between the source and destination ones, he would have no way of getting the information needed.

In the status quo, network management executes within the administrative bounds of the network manager. Once we travel outside the bounds, the network manager is often unable to obtain useful information due to lack of knowledge on where information lies. However, as seen in Bob's example, applications and users are not limited by domains like network management is, so the management problems they face span multiple administrative and management domains. Thus, it would make sense to expand network management to be able to navigate through multiple domains. The problem is twofold:

1. There is no good way for the network manager to know where to find the information it needs across many domains.
2. There is no good way to figure out access privileges for information, i.e. how much information to make available to share and who can get which information.

It would be nice, then, if Company A's network manager had a method that solved these problems to diagnose Bob's problem.

1.2 Problem with Current Networking

The current network architecture was built in the 1960s where there were a handful of machines in the world. At the time, it was important to have a small number of machines be able to

communicate to each other. Consequently, networking was built on the idea that most of the interaction over a network would be for one machine to talk to another specified machine. Getting information from one endpoint to another endpoint was the center of this design. However, the network has changed since then. In current times, with the massive influx of digital material and the change in how people use the network, machines spend more than 99% of the time obtaining information [5]. This calls for a desire to find a new network architecture that focuses on information-sharing rather than nodes in a network.

1.3 Information-Sharing for Network Management

To solve the problem in our scenario in Section 1.1, Company A needs a framework or architecture for distributed, multi-domain network management. There are two different sides to this problem: computation and the information over which computation occurs. We concentrate on the information over which computation occurs. Rather than sending data to and from two known locations, the important factors to the information-centric space are finding, accessing, and retrieving the piece of data. Physical locations are unimportant in this space. One reason is because the data could be copied to many different locations for failure tolerance or load balancing, so we would not care which copy of the data we get. If a copy fails, knowing the physical location of the failed copy would be useless. We need only be redirected to another copy. Another reason is because not everything we find interesting in the network actually has "a location". For example, a path is composed of many network nodes and the links between them. We may be monitoring the behavior of a whole path for how well it is behaving, but the location where we store that information is not obvious. Information-sharing described in Section 1.2 fits perfectly with our focus on information for network management. Using information-sharing solutions, we apply it to network management to achieve our goal of finding information for network diagnosis.

Our study focused specifically on the scalability aspect of an information-sharing or information-centric network management architecture. We believe that scalability is important in finding an information-sharing solution because network management is a world where a great amount information comes in daily across the network. In order to be able to keep track of all this information, scalability is a must in the architecture we wish for.

Of the current information-centric architectures available, we chose one called the Publish-Subscribe Internet Routing Paradigm (PSIRP) to use for network management. We chose PSIRP because of its potential in scaling. We analyzed this specific aspect of PSIRP to see if we could use it for our network management problem. Our findings showed that while PSIRP has some problems scaling because of its internal structure and lack of a global reference point, its use of Chord still allows it to achieve an $O(\log^2 N)$ tradeoff between performance and distribution.

1.4 Roadmap

The thesis will be organized in the following way. Chapter 2 talks about related work. Chapter 3 gives a summary of Chord, a protocol important to PSIRP. Chapter 4 then examines in detail PSIRP's design. Chapter 5 analyzes Chord's scalability, while Chapter 6 analyzes PSIRP's scalability. Chapter 7 is the conclusion.

2 Related Work

There are two parts of the story to look at in related works. Because we are focusing on information-sharing specifically in the network management space, we need to study previous works in this area. At the same time, since we are using an existing information-centric network architecture to solve our network management problem, we also look at works in this second area. This chapter is split into these two points.

2.1 Sharing In Network Management

Note that on the higher level, we are focusing on network management problems that need information from external domains. While domains may be reluctant about sharing their network's information to other domains, there are three main reasons why there is significant value to sharing [11]. The first reason is for distributed applications. Suppose there is an application that is running on multiple machines all spread amongst different network domains. The machines themselves are in static locations, so in order to combine results they will need to share information across the domains. The more sharing allowed, the more efficiently the application will run.

The second reason why sharing is useful is that it benefits mobile users. Just like Bob in our example in Chapter 1, there are a lot of users who use remote networks. If a problem arises, it is natural for them to ask assistance from their local network managers due to familiarity reasons. In this case, the local network administrators will need to know some information about the remote network domain in order to help the customer.

The third reason to share information is because it will help quicken the discovery of malicious activities. When we look at low volume zero day attacks or anomalies, we find that the first attack is always at the end nodes. However, the end nodes cannot tell whether or not

abnormal traffic is malicious because they need aggregate information from other nodes to discover attacks.

From the three reasons stated above, we can see the importance of sharing information in cases ranging from static to mobile users. However, it is also important to note that fully-open sharing will never be possible. There are proprietary data that will always be hidden. In addition, there are also issues of privacy, trust, and security that will prevent data to be fully open. Moreover, if all information was open to the public, efficiency would drop greatly because of all the traffic caused by unnecessary information floating around.

2.2 Information-centric Networking

Currently, large networks are broken down into small, local sub-networks called domains. There are two different types of routing that revolve around domains: intra-domain routing and inter-domain routing. Nodes within a domain communicate with each other using intra-domain routing. This is because intra-domain routing calculates detailed path heuristics to determine which route to take when routing from start node to destination. For example, Open Shortest Path First (OSPF), an intra-domain routing protocol, selects the shortest path from start node to destination for routing [9]. In order to maintain OSPF paths, updates must be made to the routing table whenever the domain adds or removes nodes or when network links are down. If a large network used OSPF, updates would be slow and the routing table would be often out of date. This is why large networks are split into numerous different domains connected to each other by some border routers. Since intra-domain routing does not scale well, another type of domain is needed to route between these different domains: inter-domain routing.

Inter-domain routing is needed because it is able to handle scaling issues. The current solution uses the Border Gateway Protocol (BGP). BGP consists of Autonomous Systems (ASes),

or domains, that communicate to each other via border routers [2]. These border routers contain routing tables to neighboring border routers and the paths they serve along with the number of AS hops. In order to pick between different AS paths, the border router looks at the number of hops as a metric for efficiency.

While the current model is able to use BGP for its inter-domain routing, information-centric networking requires a new inter-domain routing protocol to fit the new model. There are a handful of existing projects that explore this space of information-centric networking. We give a brief summary of each project in this chapter and go into more detail of the PSIRP project in later chapters. Notice that while each of the projects have unique terms for certain actions in the design, all of the projects follow the publish-subscribe service model [3]. In this model, users subscribe to topics they want information about. Publishers then publish to these topics. Once some information is published, the model sends the published data to all the subscribers of that topic.

2.2.1 DONA

The Data-Oriented Network Architecture (DONA) is an architecture that can run over the current network [7,8]. An object in DONA is associated with a principal. The name of an object, P:L, is made of the cryptographic hash of its principal's public key (P), and a unique label that the principal chooses for the object. Objects of name P:L can only be served from hosts that principal P had given permission to.

There are two actions that a node can take in DONA-- FIND(P:L) and REGISTER(P:L). The former finds information P:L and sends it back to the asking node, while the latter registers a piece of data P:L into the network. In other words, FIND is a subscribe request while

REGISTER is a publish request. Every domain in DONA contains a resolution handler (RH). These RHs, like the domains, are organized hierarchically. The RH is in charge of maintaining the registration table, which is similar to a routing table. The table consists of mappings of data names to the next-hop RH and the distance to that copy of the data. Every client knows the location of its local RH from some local configuration. When a client looks for something, it sends a FIND message to its local RH. The RH then looks into its registration table to find the longest-matching prefix to the node's request. If it cannot find P:L, it looks for an entry for P:*. If an entry exists, the RH will pass the request along to the next-hop RH as indicated in the table. However, if an entry does not exist, the RH will pass the request to its parent RH in hopes that its parent will have more information. If no lead is found there, the FIND keeps getting passed up the tree until it hits the top tier RH. Once the copy of P:L is found, though, DONA uses standard IP routing and forwarding protocols to send the data back to the source of the FIND command. If P:L cannot be found even at the top tier, an error message is sent back to the user.

When a node wants to publish some piece of information, it sends a REGISTER message to its local RH. When an RH receives a REGISTER, it first authenticates the source of the message and adds it to its registration table. It then forwards the message to neighboring RHs if P:L is a new entry in the registration table or if P:L is closer than the previous P:L copy. In other words, it will forward a REGISTER message if it will require the other RHs to update their registration tables.

An advantage to DONA is that the names are human-friendly. Thus, each node can have its own mapping from a P:L to its human-friendly name. This means that if a network decided to give new names to everything, only the mapping from human-friendly names to P:Ls needs to change, making the system more flexible and extendable. However, the disadvantage to DONA

lies in its flat names. Because objects names are flat, DONA is unable to scale to a large quantity of objects.

2.2.2 CCN

Content Centric Networking (CCN) uses faces, a generalization of interfaces, to send packets to their destinations [6]. Packets come in two types – Interest packet and Data packet. An Interest packet expresses a request for a certain piece of content, while a Data packet contains the actual content of some data. The naming in CCN is hierarchical and at least partially meaningful to humans. Because of the hierarchical structures, nodes can match longest prefixes when passing packets instead of matching the whole name. Additionally, CCN uses three data structures – Forwarding Information Base (FIB), Pending Interest Table (PIT), and Content Store (CS) – to provide caching and to forward packets. In order to understand CCN, we will discuss these structures in order.

The FIB is a table that tells the node a list of faces that point in the correct direction of an Interest. This way, when a node receives an Interest whose matching Data is not in the node's cache, the node can look up the entry in the FIB that corresponds to the Interest. The lookup results in a list of faces. The node can then try to forward the Interest to one face in the list and expect a Data packet to come back from that face. To set up the FIB, nodes that serve content periodically broadcast an advertisement to their neighbors about the Interests that they can serve. This is similar to publishing information. When a node hears of such an advertisement, it associates the advertisement's Interest with the face from which the advertisement came from. Next, it puts the information into the node's FIB. The node then broadcasts to its neighbors, letting them know that the node knows a way of getting that particular Interest.

The PIT is a table of source faces for unsatisfied Interests. Nodes can determine whether a piece of Data is unsolicited by looking into their PITs. If a node knows where to obtain Data for a certain Interest, it does two things: 1) the node sends off the Interest out to the face that can serve the Interest, 2) the node saves the information of the Interest and the face from which it came inside the PIT. Thus, if a node receives Data whose matching Interest is not in the PIT, it knows that there was no prior node that asked for the information. Interest packets are first created and sent from clients. When a client wants to know about a certain piece of data, the client sends out an Interest packet for that data. In this way, the Interest packet is similar to a subscribe event.

The CS provides LRU caching of data. As mentioned earlier, when a client has interest for some content, it first broadcasts its Interest for the content to its neighbors. The Interest trickles through the network until it reaches a) the publisher node containing Data that matches the name of the Interest packet, or b) a node that has the matching Data in its CS. The node with the Data packet sends out the Data to the incoming face. Whenever desired Data travels through the network back to its subscriber(s), each node that the data touches also caches the information in its CS. As a result, the next time the requester shows interest in the data, a closer node caching the data can quickly send its cached copy to the requester.

An advantage to CCN is that the concept of locations can completely be discarded. However, a disadvantage lies in the rigid hierarchical structure of data names. Another disadvantage is in the initial distribution of a piece of Data. Because the information that a node serves a certain piece of Data must be broadcasted over the entire network for all nodes to know about that information, scalability becomes a big issue.

2.2.3. NetInf

Unlike CCN which passes around information like breadcrumbs, NetInf has a systematic method for finding a piece of data and retrieving it [1]. In NetInf, the information model consists of two objects – Information Object (IO) and Bit-level Object (BO). The BO contains either information of the desired object, called the BO handle, or a pointer to another IO. An IO contains the object ID, metadata, and the BO. The ID can be globally unique and acts as the name of an object. The ID is also not dependent on the storage location, so it stays consistent regardless of its source. The metadata in an IO includes the IO's attributes. To put IOs onto the network, publishers publish them much like the publish-subscribe model.

In order to find a piece of information, NetInf uses a request/resolve model. The client first requests IOs that have the attributes that the client is interested in. This is similar to the subscribe action. The NetInf resolution service (NRS) uses the attributes to find matching IOs to return to the client. The client then selects an IO of interest from the list of returned IOs and receives the content via the BO field inside the IO. Every NetInf node caches all BOs that come its way to increase efficiency.

Name resolution is a large part of NetInf, and the key to making this scalable is to split the network into zones. Each zone can have its own method of name resolution. For example, local broadcasting zones can use broadcasting for name resolution as opposed to larger, more complicated zones that use more advanced routing protocols like Multiple DHTs (MDHT) or Late Locator Construction (LLC). In MDHT, names are resolved through dictionary nodes (DNs) in the network. DNs contain mappings from object IDs to other IDs or addresses. The DHTs are organized into about four different levels depending on the type of node information it

contains, and the nodes are hierarchically connected. A higher-level DHT contains everything below it. In LLC, the system adapts to mobile environments by constructing addresses on the go.

The main disadvantage to NetInf is scalability in MDHT. If the top level DHTs contain everything below it, the number of entries in the DHT would be extremely large.

2.2.5. PSIRP

The Publish-Subscribe Internet Routing Paradigm (PSIRP) project details an Inter-domain Rendezvous Service Architecture that allows a node to find information across the network [10]. In this system, every object is named by a rendezvous identifier (RId). Each RId belongs to one or more scopes. Scopes are special objects that contain other objects (including other scopes), so they themselves also have unique identifiers. These scopes represent categories that objects fall under. For example, a packet trace from Boston, MA, can be put into the scope "packet trace," "Boston," "MA," "USA," etc. There is an assumption that every scope is contained in a root scope so that one can always go to the root scope to find some information. While RIds in each scope are statistically unique, there is no guarantee that the RIds are globally unique. A workaround to this issue is to construct RIds such that they include the full paths from root scope to local scope, making them globally unique.

The PSIRP network is set up as a connection of hierarchical overlays. The local portion of an overlay is organized into a Canon structure [4]. The global portion of the overlay connects the Canon hierarchies together. When a RId is not found locally, the query is passed to the rendezvous node in a rendezvous network. This node communicates with other rendezvous networks to find the location of the RId. At the global level, every node belongs to a prefix group which contains all nodes that share the same prefix. Ever node also stores direct links to every

node in its prefix group, as well as at least one link to another prefix group. In a sense these prefix groups are connected in its own Canon structure. This serves to speed up the process of routing. However, though PSIRP uses Canon, it is not taking full advantage of the design because the application does not apply to PSIRP. Canon allows DHTs to easily merge together as virtually larger DHTs that contain all nodes in the children DHT. If PSIRP's prefix groups employ Canon's merging, the top layer would contain all the nodes in the inter-domain overlays.

The main issue that PSIRP faces is the issue of scalability. Because a piece of information can belong in an unlimited amount of scopes, finding something in the network can take a long time. While normal tree structures have the most nodes at the leaves, the large number of top-layer scopes can lead to an inverted tree structure where there are more nodes at the top layers than at the leaves.

Because of the existence of scopes as a core element of the architecture of PSIRP, PSIRP is our choice at present for the basis of our information substrate for inter-domain network management. Hence, the focus of this thesis is examining the scalability of inter-domain routing in PSIRP. In Chapter 4 we will examine PSIRP in more depth.

3 Summary of Chord

Because PSIRP uses Chord as an integral part of its design (see Chapter 4.2), it is important to first have an understanding of Chord before moving on to PSIRP. Chord is a protocol for building, searching, and deleting objects in a peer-to-peer distributed hash table (DHT) [12]. Its methods for maintaining the DHT allow a lookup time of $O(logN)$ and require only $O(\log^2 N)$ number of messages to keep the system updated when a node joins or leave the network, where N is the number of nodes in the network.

3.1 Setup

A Chord system is a network of no more than 2^m nodes, where m is the number of bits in a fixed ID space. This means that a Chord system's ID space ranges from $[0, 2^m - 1]$. Each node is assigned an ID in the range $[0, 2^m - 1]$. The nodes and keys are then put through a consistent hashing function, such as SHA-1, to obtain an identifier for each of them. These identifiers determine where the node is placed inside the circle. The identifiers also determine which key is in which node. A node, then, is responsible for a sub-range of keys in the ID space. Because of consistent hashing properties, identifiers for nodes and keys are selected pseudo-randomly. This results in an evenly-spread load amongst the nodes.

A node's successor is the next node in the circle moving in a clockwise direction. In other words, the successor node of a key, denoted $successor(k)$ where k is the key, is the node whose identifier is either equal to or after the identifier of k. The predecessor is the opposite of successor. A predecessor is the closes node in the counter-clockwise direction. Figure 1 shows the layout of the network. A key with identifier 1 would fall into node 1. A key with identifier 3 will fall into node 5 because there is no node at 3 and node 5 is the next one available.

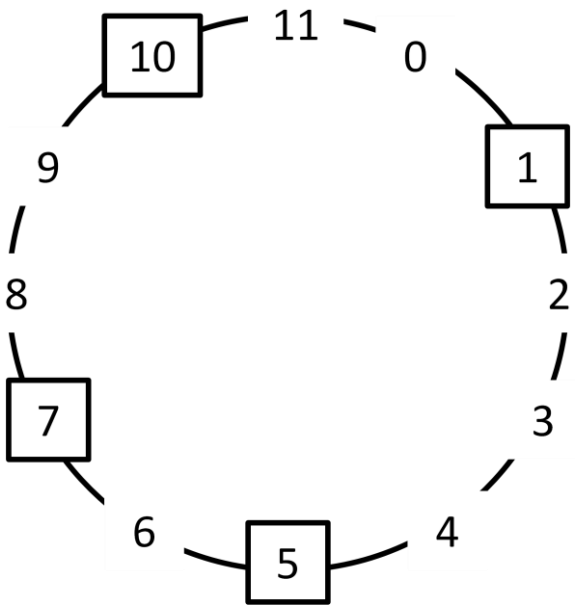


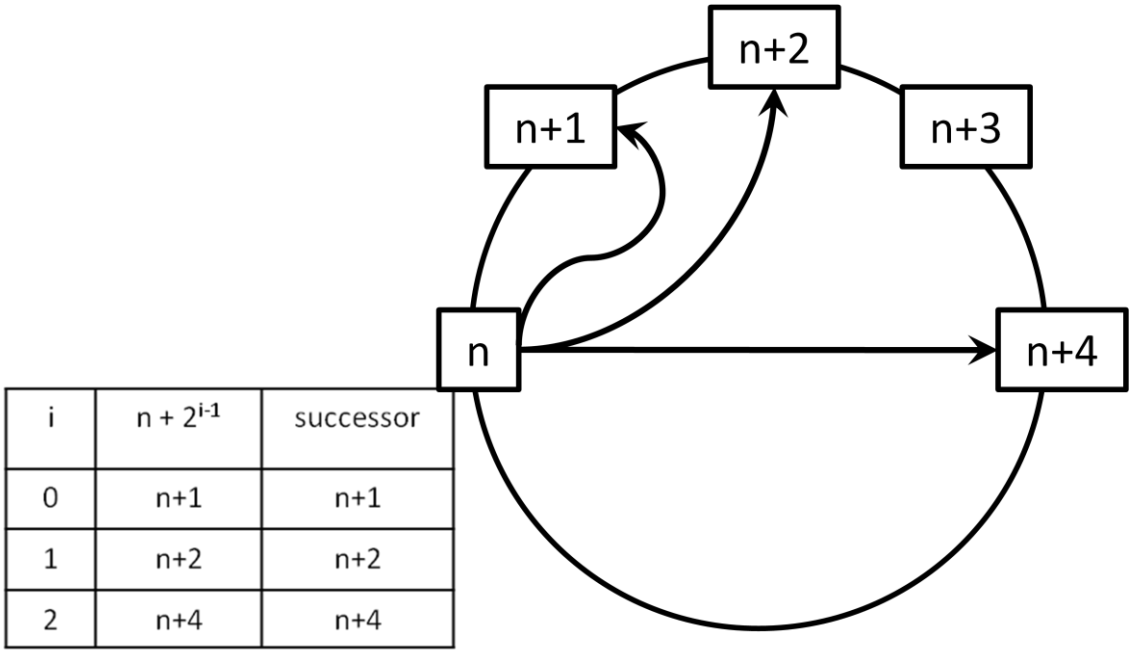
Figure 1. A Chord circle layout. The numbers with rectangles are joined nodes and their identifiers in the system, while numbers without rectangles are placeholder identifiers for future possible nodes.

i	$1 + 2^{i-1}$	successor
0	2	5
1	3	5
2	5	5
3	7	7

Table 1. The finger table for node 1 in from Figure 1. Note that because there are no joined nodes at points 2, 3, and 4, the successor for those points is node 5.

When Chord inserts keys, it inserts it such that the key is put in the node with identifier equaling $successor(k)$. In order to find keys across the network on other nodes other than itself, a node n keeps something called a finger table. This table contains up to m entries, where the ith entry corresponds to the immediate node that succeeds or equals $(n + 2^{i-1}) mod\ m$. In other words, the ith entry points to $successor(2^{i-1})$. The equation ensures that a node is able to span half the Chord circle. Figure 2 shows the finger table of a node n in a Chord system. It also nicely shows that the finger table of a node only extends to half the circle. To demonstrate this

point, we've simplified the system to contain all nodes from n to $n+4$. For this reason, the finger table entries $n+2^{i-1}$ and $successor$ are identical. However, note that the two entries may not always be identical. An example for this case is shown for the finger table for node 1 (Table 1).



i	$n + 2^{i-1}$	successor
0	n+1	n+1
1	n+2	n+2
2	n+4	n+4

Figure 2. A node's finger table and where it points to on the Chord circle. In this circle, all nodes from n to n+4 have joined the circle. This assumption is made to visually display that in a perfect scenario, the nodes inside a finger table cover exactly half the circle, then a fourth, then an eight, etc.

3.2 Lookup

With finger tables set up, the lookup procedure of Chord is straightforward. When a query for a key comes in, a node looks to see if it is in charge of the key. If not, it looks inside its finger table and forwards the query to the largest successor in its finger table that does not exceed the key's id. The process repeats until the key is found, at which point the data is sent back to the originating node. By having the node forward to its largest successor not exceeding the key, Chord cuts down the search size of the DHT dramatically. At worst case in which the key is farthest away

from the node, the search path still decreases by half after every iteration of the search (Figure 3). As a result, the lookup time for Chord is only $O(logN)$, where N is the number of nodes.

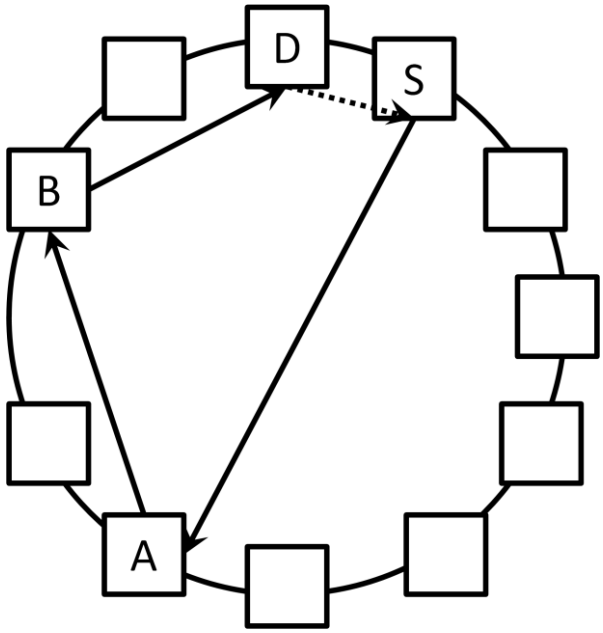


Figure 3. The lookup path from node S to D. S first forwards to A because D is not closer than A and A is the farthest S can go before going over the half circle mark. Similarly, A forwards to B because the next option would have been S, which would have overshot the target. Lastly, B forwards the request to D. D then directly sends the data back to S because the query contains the location of the source.

3.3 Advantages and Disadvantages

There are many benefits to using Chord. One such benefit is Chord's fast lookup time of $O(logN)$. Another good thing about Chord is that nodes can easily join and leave. Remember that each node handles a sub-range of the ID space. When a node becomes overloaded, a second node can join the network and take half of the sub-range that the overloaded node handled. This event is very localized because it only affects the overloaded node, the joining node, and a handful of other nodes whose finger tables need to update as a result of the new node. Similarly, if a node wants to leave the system, neighboring nodes take over the ID space the leaving node covers. Again, only a handful of nodes need to update their information to accommodate the

change. The total number of messages needed to update nodes when a join or leave happens is only $O(\log^2 N)$.

Because of consistent hashing, nodes can join and leave while neighboring nodes update themselves to hand over data or swallow the unclaimed data. This allows for local, quiet joins and leaves. Both of these benefits are advantageous to scaling in Chord. With a fast lookup and the flexibility to add nodes, it is easy to add more to a Chord system and have it maintain its performance.

The downside to Chord, as with all single DHTs, though, is that there will come a point when there are just too many nodes to handle. Even if Chord is well-adapted to scalability, a single Chord system was not designed to be able to handle all the data on the internet. Similarly, it was not designed to hold every piece of information needed for network management. Thus, in Chapter 5 we explore nested Chord systems and its efficiency at dealing with large scale data that will be needed for an information-centric network management PSIRP system.

4 A More Detailed Look into PSIRP

PSIRP uses a publish-subscribe paradigm to create and find information while using an inter-domain architecture for scaling purposes. In order for this to work, PSIRP employs a few invariants [13]. In this chapter we will discuss these invariants and how PSIRP uses them to achieve its model. In particular, we will focus on the rendezvous aspect of PSIRP and how it navigates through scopes.

4.1 Invariants

PSIRP uses five invariants to build the model:

1. Information objects are identified with self-generated flat label identifiers called Rendezvous Identifiers, or RIds.
2. Scopes are information objects with special properties. Specifically, scopes are used to hierarchically categorize objects. They may contain both information objects and other scopes (Figure 4). Each information object belongs to at least one scope. One scope does not need to belong to another scope, but it can also belong to one or more scopes. The scopes also are identified with flat label identifiers called Scope identifiers, or SIds. Both RIds and SIds are statistically unique within the scopes they are in. In Figure 4, the leftmost information object has the identifiers $/SId1/SId3/RId1$ and $/SId1/SId4/RId1$. Each system has its own root scope. While the root scope is not explicitly identified, it is understood that there is such a scope that encapsulates all the top level scopes inside the system. This is represented by each object's fully qualified identifier

having a leading slash in its names.

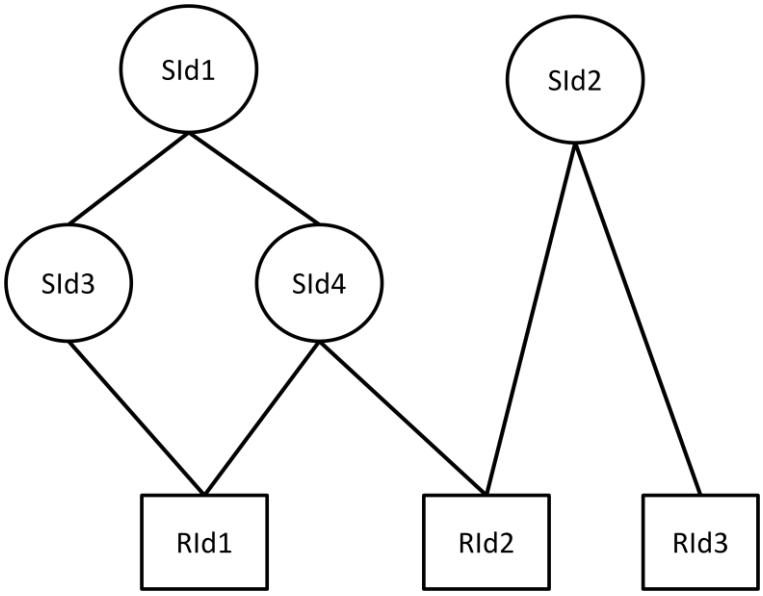


Figure 4. A possible layout of scopes and information objects. The leftmost item has the identifiers /SId1/SId3/RId1 and /SId1/SId4/RId1. Similarly, the rightmost item has the identifier /SId2/RId3.

3. PSIRP uses a publish-subscribe service model. Scopes are published into the root scope or sub-scopes. Subscribers may subscribe to any scope they desire. By subscribing to a scope, users get notified when an object gets added in the scope. When a scope is published, subscribers to that scope's parent scope are notified of the event. Unpublishing a scope involves removing all references to that scope. A publisher advertises information into scopes. This essentially places an information object into a scope. A subscriber then subscribes to a scope or information and can subsequently unsubscribe. Finally, when the publisher wants to publish some data, the publish action publishes the data to the subscribers of that information.
4. Disseminating information in a scope is separated into three functions – rendezvous, topology management and formation, and forwarding. Rendezvous is the process of locating the information desired by matching publisher information and subscriber

interests. The result is then passed on to the topology management and formation function, which finds a topology to relay the information to the subscriber. Forwarding is then used to physically send the information to the subscriber. The behavior of these functions is determined by the last invariant, the dissemination strategy.

5. Each scope has a dissemination strategy. This defines how the scope transfers information.

4.2 Rendezvous

As explained above, when the system needs to match publisher information and subscriber interests, the rendezvous function is called. The Rendezvous function varies depending on the dissemination strategy involved. For example, for dissemination within a scope, the rendezvous function simply follows the hierarchical path of the information object's identifier to locate the object and disseminates the object. However, when a scope is looking for something outside its scope, the Rendezvous function may need to use its Rendezvous Point (RP). Each scope belongs to one RP. In order to find some information, the scope asks the RP. The RP then locates the scope containing the matching information needed.

The rendezvous function is important because it is the stage we focus on when analyzing PSIRP. In order to match information to interests, this function essentially needs to search through many scopes to find correct scope. To help the search process, the scopes are organized into Canon DHTs for faster lookups. Because PSIRP uses its Canon DHTs like Chord DHTs instead of taking advantage of Canon, we treat it as if PSIRP were using Chord.

5 Analysis on Chord

The method PSIRP uses to find a needle in a haystack is the Chord system and its way of finding information. As described in Chapter 3, Chord provides quick lookup by drastically cutting down the search space every time it hops to a different server on the system. Chord can also handle some amount of scaling because the process of adding servers is localized, making it easy for the system to grow. Because of such advantages, it is important to study the use of Chord specific to PSIRP and the effects scaling has on it. Because PSIRP must handle an enormous amount of data, we looked at nested Chord systems and compared their scalabilities to that of the normal Chord system. In a nested Chord system, the outermost layer is the only layer that contains the needle. The inner layers all contain entries that direct the system to a Chord ring (i.e., one single Chord entity) in the next layer. Each layer consists of multiple Chord rings, and each server in the previous layer points to only one Chord ring in the next layer. In this fashion, a nested, layered Chord system is formed.

In the next sections, we analyze the runtimes for layered Chord systems and discuss the results as well as conclusions we can draw from them.

5.1 A Simple Nested Chord System

We examined the effects of the number of servers in each layer of a Chord system on the number of hops and overall runtime of the Chord system. To get a better understanding of how layered Chord systems work, we chose the example of having a system where one entry represented one person in the world. All the calculations were made in powers of two for simplification. Four billion people in the world meant we needed to have a system that stored about 2^{32} entries. We also assumed each Chord ring to have a fixed size of 2^{8} entries. To help with finding orders of

times, we set a variable $n = 8$. Lastly, we assumed that the runtime for one network hop was a constant factor C. To summarize, in our setup we had the following:

- $2^{32} = 2^{4n}$ entries per system
- $2^8 = 2^n$ entries per server
- One network hop costs a constant factor C
- Problem: With fixed number of entries per system and entries per server, vary the number of number of servers in each layer to find the number of layers needed and the optimal runtime.

To understand the next few figures, use Figure 5 as the legend. Note that the next few figures are only brief sketches of Chord systems and do not encompass all the rings and servers in the system.

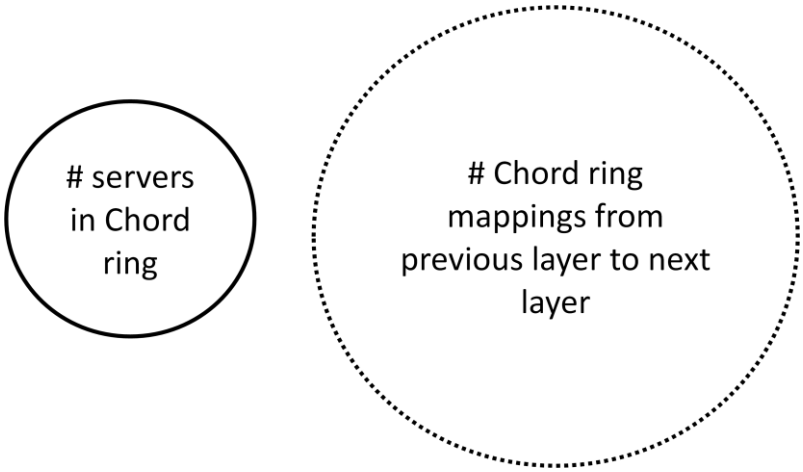


Figure 5. Solid line represents a Chord ring, while dotted line represents a Chord layer

First, we looked at the extreme case where all 2^{4n} entries were in one single Chord ring (Figure 6). In this case, the total number of servers in the whole system was 2^{3n}. All of the entries were directly in the single ring and no entries were indirections to other Chord rings. As a

reminder, the number of hops in a Chord system is $O(lgm)$, where m is the number of servers in the system. Thus, the number of hops in this case was $O(lg2^{3n})$. Because there were no indirections, the lookup time was $O(lg2^8)$ for the number of entries in one server. Combining hops and lookup, the runtime for a single-layer system was $O(3n)$. The total number of servers was $2^{24} = 2^{3n}$.

Figure 6. The Chord system with every entry in one ring.

Second, we looked at the case where the Chord system contained two layers (Figure 7). With two layers, the first layer (the innermost layer) was dedicated solely to indirection to the second layer. Like the single layer case, the second layer needed to contain 2^{24} servers in total. As a result, the first layer had 2^8 servers each containing the fixed number of 2^8 entries. However, unlike the single layer case, these servers were not all in the same ring because each server in the first layer pointed to a different Chord ring. This led to 2^{16} Chord rings on the second layer each containing 2^8 servers. Totaling this up gave the desired 2^{24} servers in the top layer. Using these numbers, we found that the number of hops in each layer was $O(lg2^8)$, or $O(n)$. As expected, this is less than the number of hops in a single Chord ring system as mentioned earlier because the number of servers in a ring decreased. Each layer also had a lookup time of $O(n)$, which made each layer have the runtime of $O(2n)$. With the added second layer, the total number of hops and lookup time became $O(2n) + O(2n) = O(2n)$. The extra network hop also added $O(C)$ into the runtime equation. The final runtime became $O(2n + C)$. The total number of servers was $2^{24} + 2^8$.

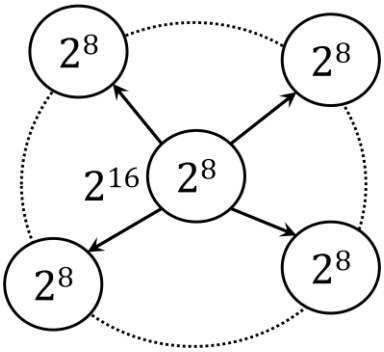


Figure 7. The Chord system with two layers.

Last, we looked at the other extreme case where there was only one server per Chord ring (Figure 8). This case required all the time to be spent hopping between servers in different Chord rings and systems. The only thing that happened at each layer was the lookup into that layer's single server for the next Chord ring to go to. The runtime for such a system was $O(n + C)$. The total number of servers was $1 + 2^8 + 2^{16} + 2^{24}$.

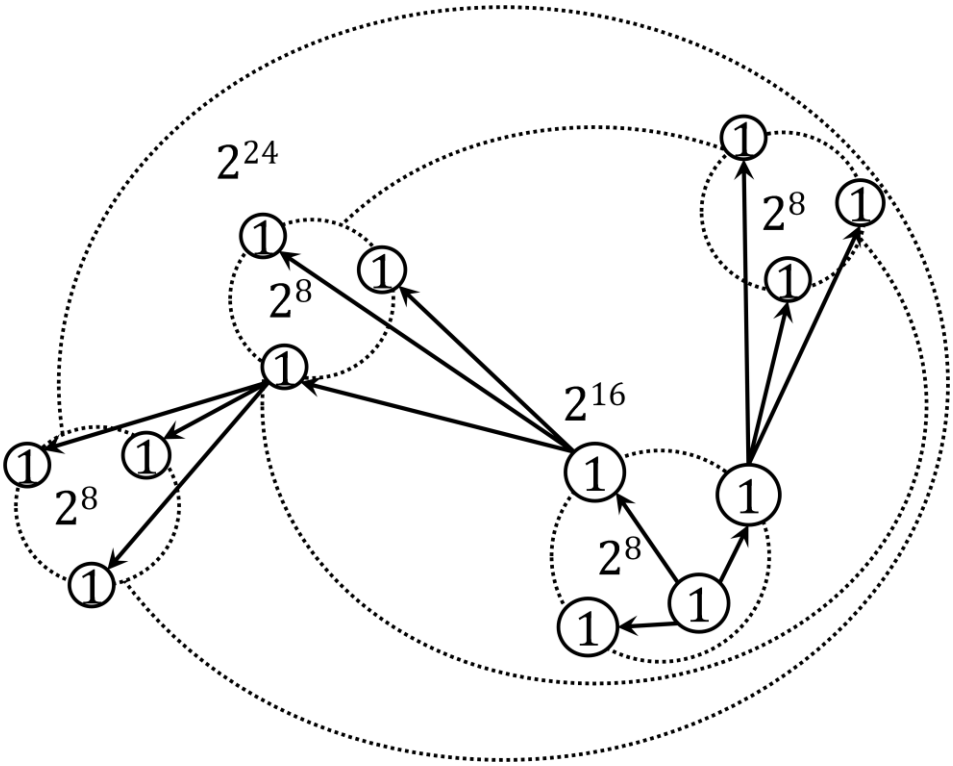


Figure 8. The Chord system with only 1 server per ring

We present our findings for the runtimes and number of servers in the three cases in Table 2. The table suggests that if we only look at runtimes, the optimal Chord system may be one where each Chord ring only has one server, which essentially strips the Chord system of its advantage in speedy lookups. A point to note, also, is that while runtime improves as we get more layers, the number of servers needed to support the whole system increases.

	One Chord Ring	Two Layers	One Server Per Chord Ring
Runtime	$O(3n)$	$O(2n + C)$	$O(n + C)$
# Total Servers	2^{3n}	$2^{3n} + 2^{n}$	$1 + 2^{n} + 2^{2n} + 2^{3n}$

Table 2. Summary of runtime for the three cases described previously, where n = 8

5.2 Nested Chord Systems on a Bigger Scale

Because looking at the three above cases was limiting, we decided to examine further what was going on by making calculations on a larger scale. Instead of having a system with only 2^{32} entries, we looked at two different systems with 2^{128} and 2^{256} entries. Each server still had 2^{8} entries. Just like what we did previously, we changed the number of servers inside a Chord ring while keeping other variables constant to find number of layers needed for the circumstance and the runtime of it. We then plotted this data to visualize the trend happening as we decreased the number of servers in a ring. To simplify runtime calculations, we ignored the constant time needed to make a network hop and instead focused on the runtime inside the rings. For each of the graphs below there will be two lines representing the two different systems. The green line will show the change for the system with 2^{256} entries, while the blue line will show the change for the system with 2^{128} entries.

Figure 9 shows the relationship between the number of servers in a Chord ring and the number of layers. As expected, the more servers there are per Chord ring the fewer layers there

are in the Chord system. We can also see that at a certain point, the number of layers levels off and stays constant no matter how much we change the number of servers in a ring.

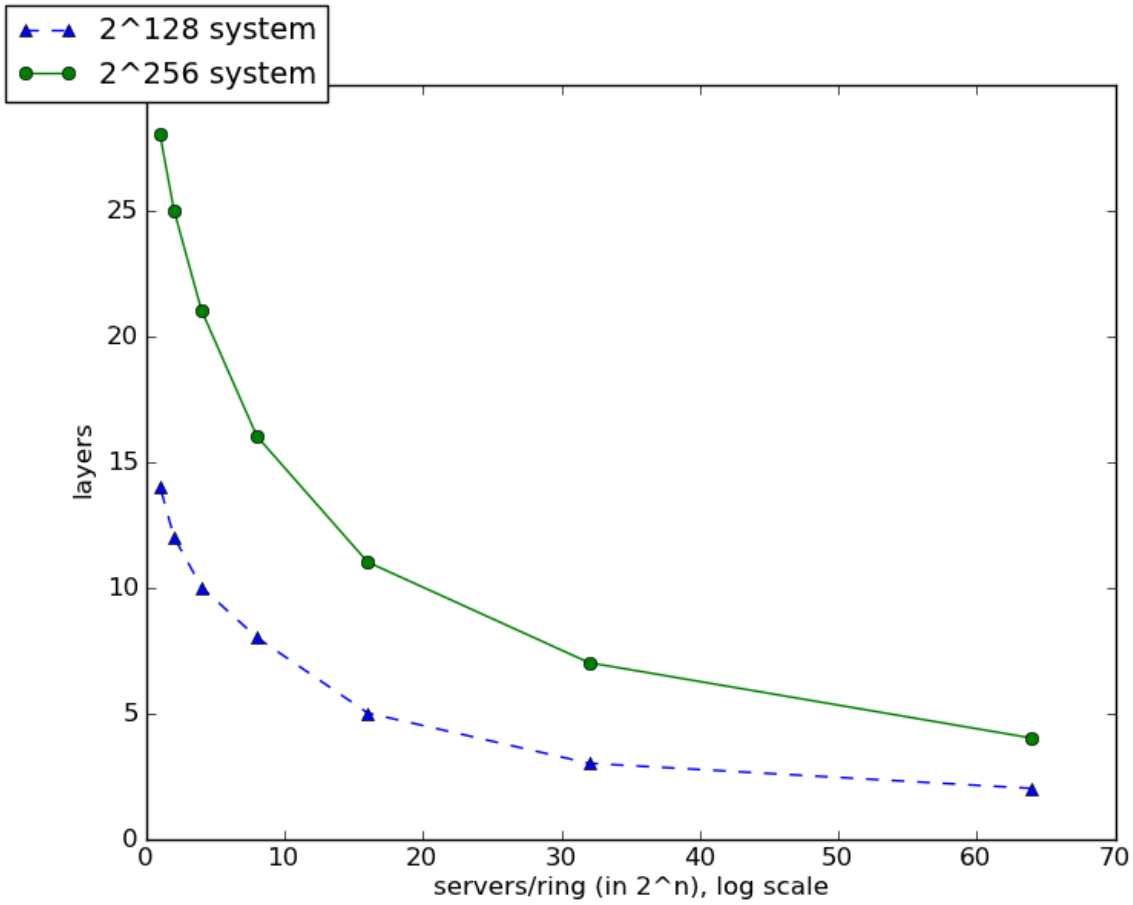


Figure 9. The graph plotting the number of servers in a ring vs. the number of layers in a system. Note that the x-axis is in log scale. In other words, the marks 10, 20, 30, etc., represent values of 2^{10}, 2^{20}, 2^{30}, etc.

Figure 10 shows the change in runtime as the number of servers per ring increases. Consistent with our findings from our previously simplified example, the runtime increases as more servers get added to a ring. This is because the system needs to spend more time searching for the correct server within a ring rather than performing lookups right away and jumping to the next layer.

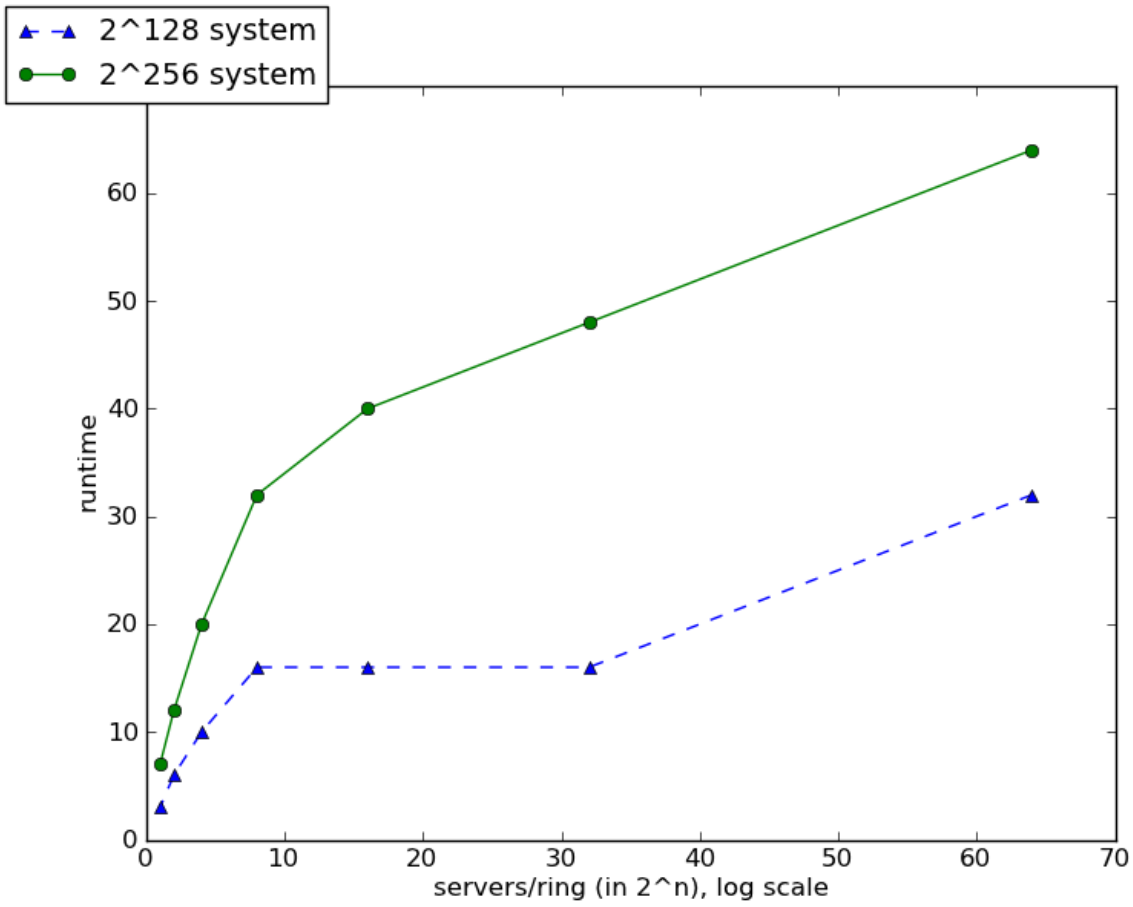


Figure 10. The graph plotting the number of servers in a ring vs. the runtime. Note that the x-axis is in log scale. In other words, the marks 10, 20, 30, etc., represent values of 2^{10}, 2^{20}, 2^{30}, etc.

Combining the graphs together, we get the plot in Figure 11. The 3D graph clearly shows us the relationships between the three variables and we are able to conclude that optimal runtime is achieved when the number of servers per ring is lowest and when the number of layers in the system is highest.

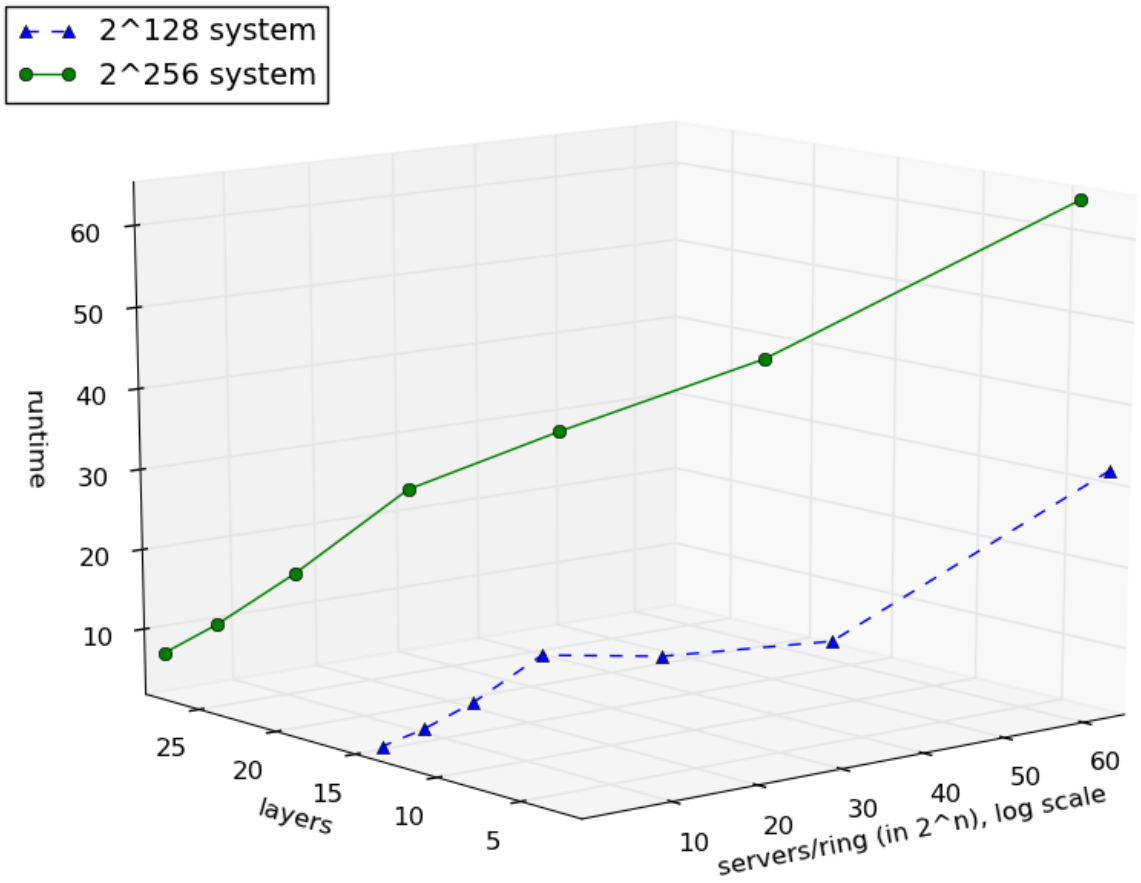


Figure 11. The 3D plot shows the relationship between servers per ring vs. number of layers vs. runtime. Note that the x-axis is in log scale. In other words, the marks 10, 20, 30, etc., represent values of 2^{10}, 2^{20}, 2^{30}, etc.

5.3 Discussion of Results

Results in Chapters 5.2 and 5.3 both suggest that it would be best if PSIRP put all its data in a nested Chord system where each ring only has one server. However, while on paper it seems this approach gives the best runtime, in actuality the structure will not provide the best results. The reason is because the more layers a system has, the less distributed its load is. In an ordinary, single-layer Chord system, the load on servers is divided because servers may take different paths to reach the same target (Figure 12). As seen in the figure, the path taken to the destination server is different in all three cases. Although the eventual target is the same, depending on the server querying the data, different servers will be asked to provide information about the location

of the final target. In this way, Chord successfully splits the load amongst the servers. This also allows for better scalability because Chord will spread the load even with an increase in the amount of activity due to scaling up. This is very useful in an information-centric network on network management because if a problem occurs over the network, many servers from different places will want to query the same information. There will be an influx of servers querying the same destination, but because Chord forces different paths onto different entry points, the network will not be overloaded along the same path. If we were to adopt a system with only one server per ring, however, we will not be able to fully utilize Chord's method of dividing the load. In addition, the large number of servers needed to handle all the layers will not scale well in the future.

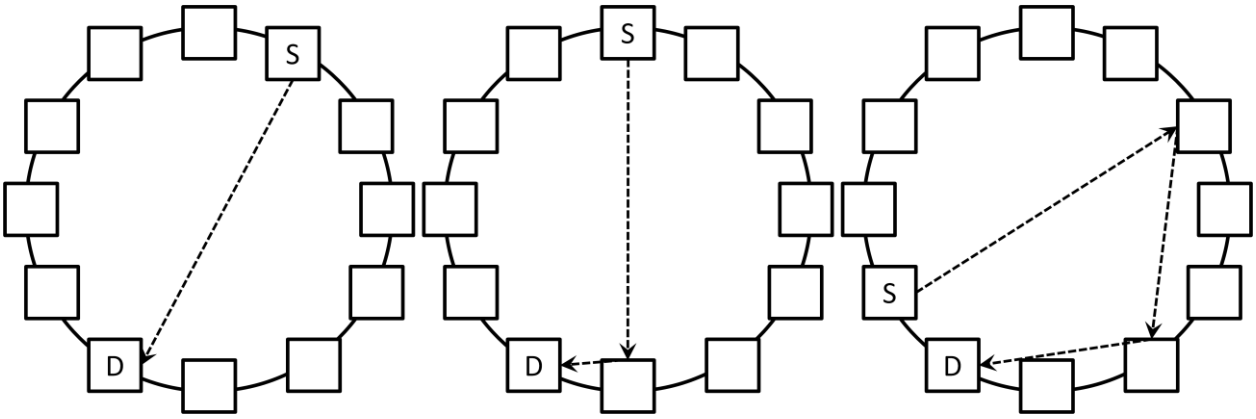


Figure 12. The three Chord systems above show the different possible paths to destination D depending on where the start server S is located.

On the other hand, it would also be impractical for PSIRP to use one Chord system. Given the large amount of data PSIRP must handle, one Chord ring will need to include a large set of servers. As a result, lookup will become slower as the number of servers increases.

5.4 Conclusion of Chord in PSIRP

In the end, we conclude that success shown through calculations on paper does not translate to success in the real world. Although Chord performance improves with decreasing numbers of servers, we also need to remember that it provides $O(\log^2 N)$ behavior in a distributed context. This would allow PSIRP to trade off in an $O(\log^2 N)$ way between distribution and performance.

6 Analysis on PSIRP's Scalability

A key aspect to note when considering network architectures to support network management is the ability to scale well. Network management handles a large amount of data, so being able to change and accommodate for the growing size is an important factor in evaluating an architecture. In this chapter, we've pinpointed three specific areas of PSIRP that we believe play roles in affecting PSIRP's ability to scale. The first area is the structure of scopes in PSIRP. The second area is the scope-joining feature in PSIRP. The last area, traversing through nodes in PSIRP, explores not a feature of PSIRP, but rather a possible extension that could improve its scalability.

6.1 PSIRP's Scope Structure

BGP is used for current inter-domain routing. As mentioned in Chapter 2, the structure of BGP is built on ASes. The ASes connect together to form a tree structure. According to how the internet is built today, this tree starts with very few ASes at the top and grows downward (Figure 13). Because of the downward-growing trend, BGP is able to scale because traversing a tree is like a standard tree traversal. If information at a particular AS cannot be found in itself or its children, it can ask the neighboring ASes above it. As the query moves up the tree, there are fewer and fewer ASes to ask.

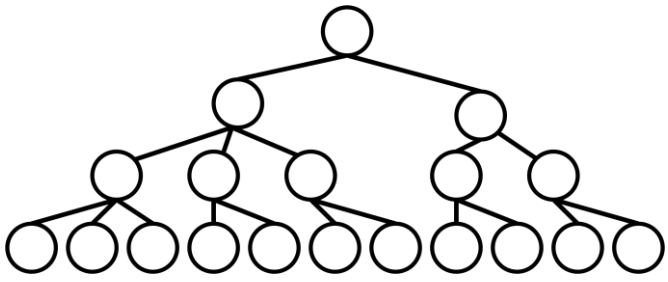


Figure 13. Structure of ASes in BGP

PSIRP borrows a key idea from BGP: using scopes instead of ASes but with a twist. As Chapter 4 describes, scopes are uniquely-identified objects that contain scopes or information objects relevant to that scope. Information objects can be in multiple scopes because they can be related to many different scope topics. This behavior is different from that of an AS, which contain information that exists only in that single AS. Because of this, we believe that PSIRP actually has a tree structure opposite of that in a BGP network – namely, the tree grows upward instead of downward. This means that there will be more scopes at the top level than there will be scopes at the bottom level, which can create a challenge to scalability.

6.1.1 Packet Trace Example in Scopes

We decided to mock up an imaginary world of a potential sub-tree from a PSIRP network. The information object being stored was a packet trace coming from MIT. As such, the object belongs to the MIT scope. Because PSIRP scopes are nested, we put MIT into two scopes judging solely on the fact that the information that MIT is a private university in Cambridge. We then started expanding the tree upward by putting scopes into potential scopes to which they belong. The tree quickly started expanding as more and more options became available (Figure 14). Remember that we only focused on two aspects of the MIT scope. In fact, there could be many more scopes that MIT falls under. Similarly, the information object does not only belong in the MIT scope. There could be scopes detailing specific network types that the packet trace could fall under. In addition, the packet trace could belong to a scope based on its specific date and time. Those scopes then in turn could belong to different days, months, years, etc. scopes. The information could also belong to a scope specifically for packet traces at MIT, which in turn could belong to a packet traces scope. The options are enormous. As you can see, one simple

packet trace itself actually falls under many scopes besides the ones in the figure. Those then belong to even more scopes, making the tree grow upwards even more.

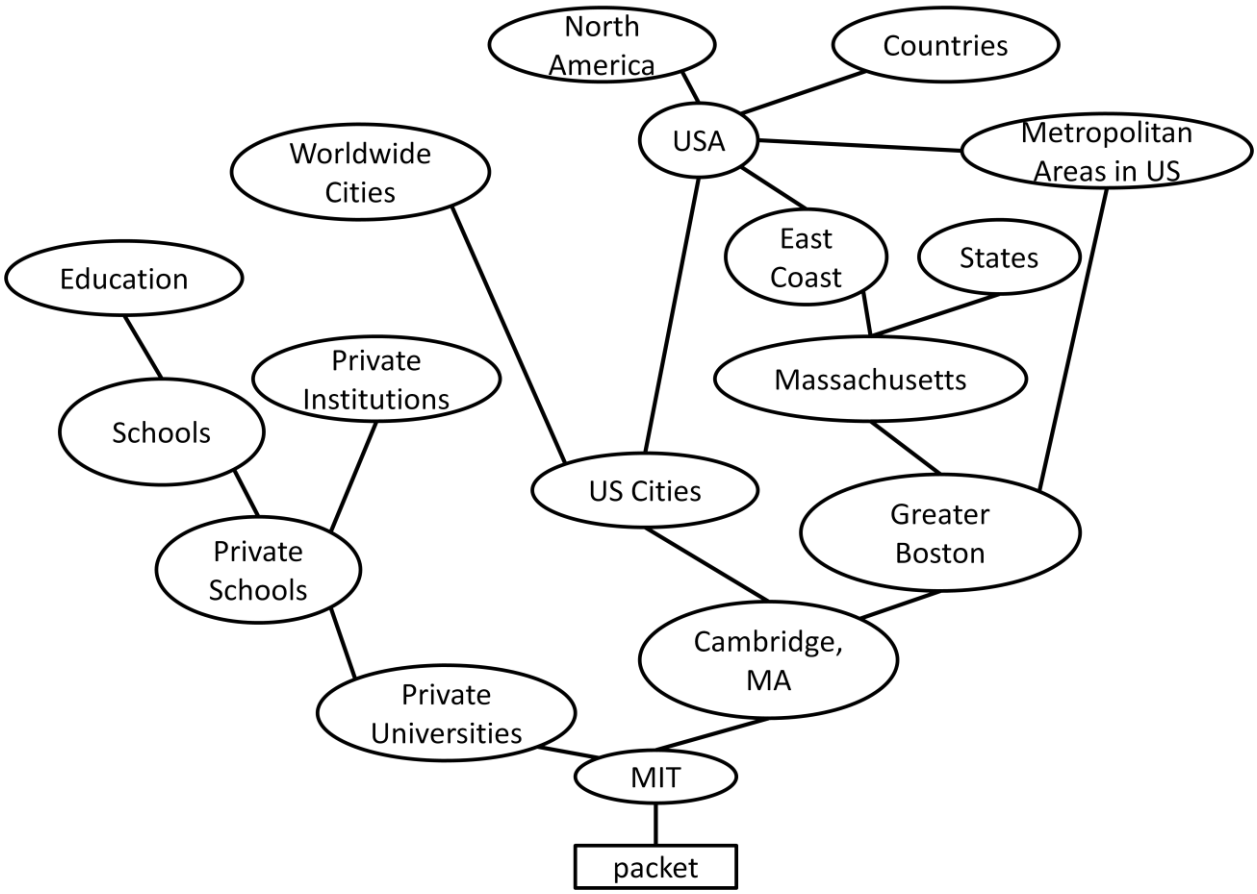


Figure 14. Structure of Scopes in PSIRP

One may argue that eventually scopes converge at the top to a single scope. This is indeed possible. As we move higher up, the tree may again start to get narrower. For example, the scope *North America* in the Figure 14 could belong to a *Continents* scope, which is as far as it goes. This possibility does not eliminate the fact that, to a certain point, the tree still expands upwards from the leaves. At best, the tree could form a structure that grows and shrinks (Figure 15). Even with a growing and shrinking tree, the system will at some point still need to go through a vast number of scopes to find the correct scope.

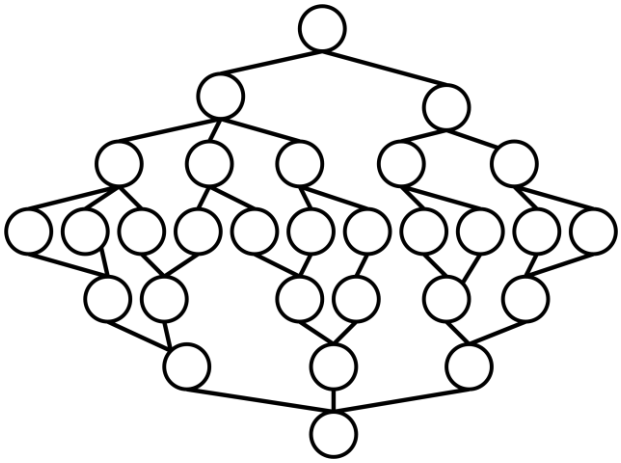


Figure 15. Revised Structure of Scopes in PSIRP

One may also argue that our example is inconclusive because it only takes into account one information object and part of its scopes instead of the complete world. In a whole world, perhaps there would be so many information objects and lower-level scopes that the tree will never exhibit any obvious signs of growth from the bottom upwards. However, we do not believe this will happen. The reason is that there is only a limited set number of information objects that are needed in network management. However, the possibilities for scopes are endless. New scopes can always be added to reorganize data or redefine relevance. For every new piece of information, there could be new scope creations used to further define the specifics for the information at hand. If a packet trace from MIT appeared on July 01, 2011 04:00:00, there could be a new scope for *July 01, 2011 04:00:00*, a new one for *July 01, 2011, 04:00:00 packet traces*, a new one for *MIT July 01, 2011, 04:00:00*, etc.

6.1.2 Impact of Scopes Structure on Scaling

Our example above showed that PSIRP scopes do not follow the conventional Internet routing tree structure. The PSIRP creators believe that their design scales based on the fact that BGP scales, but this contradicts the case where scope structures differ from AS structures. Because we

found that the scopes do not form a tree pattern similar to that of ASes, we conclude that PSIRP cannot effectively use the approach of a BGP network. In particular, the different structure gives rise to a problem in scaling.

In PSIRP, nested scopes are designed for scalability. The design was meant so that they allow the system to easily search down the tree. The key for this to work efficiently is if a tree has few scopes on top because fewer scopes on top mean fewer scopes to search through when trying to find a specific one. What we've found, though, is that PSIRP does not have this structure. Instead, as the network gets larger and larger, there will be more and more nodes the higher up we go in the tree. Recall that scopes are stored in Chord systems. Using Chord, searching through any point there is an explosion of scopes, even if it's not the top-level scopes, is difficult to handle. Thus, PSIRP's explosive structure, coupled with Chord, will make it hard to scale.

6.2 Scope-joining in PSIRP

In Chapter 4 we explained the naming convention of PSIRP. The important thing to note is that identifiers are unique within its own scope system. In other words, multiple scope systems can have the same identifiers. While this is ok if systems stay to themselves, problems quickly arise if we need to join root scopes together.

6.2.1 Examples of Scope-joining

To examine the nuances of scope-joining in PSIRP, we set up scenarios that would require two scopes to join. The two independent scopes had identical scope structures. There were three top-level scopes. Two of the top-level scopes also contained another scope. The third top-level scope

contained only one object. The two mid-tier scopes each contained one object. Next we supposed that the two scopes wanted to join.

In our first experiment, we considered a scenario where two scopes were created separately but in fact should've belonged together. Because the scopes are small, it is likely that none of the identifiers in the two scopes match each other. Thus, we set up a world where the independent scopes contained completely different full-path identifiers and noted how the joined scope would look (Figure 16). Even in the case where the two scopes had the same identifier for two different objects (in the case of $O1$ in both scopes), the objects' full paths were different ($/S1/S4/O1$ vs $/S6/S7/O1$). Notice that the root scopes of these two independent scopes were unnamed. Recall from Chapter 4 that root scopes are not explicitly identified. To join the two scopes, one simply needed to merge the unnamed root scope into the same unnamed root scope. The resulting scope still maintained the full path names for the objects.

In our second experiment, we considered a scenario where two regular-sized scopes suddenly found a point in commonality and thus wanted to join. It is likely in this scenario that some paths overlap. For simplicity sake, we set up a world where only one full-path identifier in both scopes matched each other (Figure 17). $O2$ in both scopes had the same full path, namely $/S2/S5/O2$. Even if the content of the two $O2$'s were different, PSIRP would not know that because the system only looks at identifiers to distinguish objects. When we simply merged the root scopes together like we did in the first scenario, we got a situation shown in Figure 18. The resulting scope had two paths to $/S2/S5/O2$. The problem with two paths to two objects containing different content is that when a subscriber asks for the content, there is a chance that PSIRP returns the wrong content.

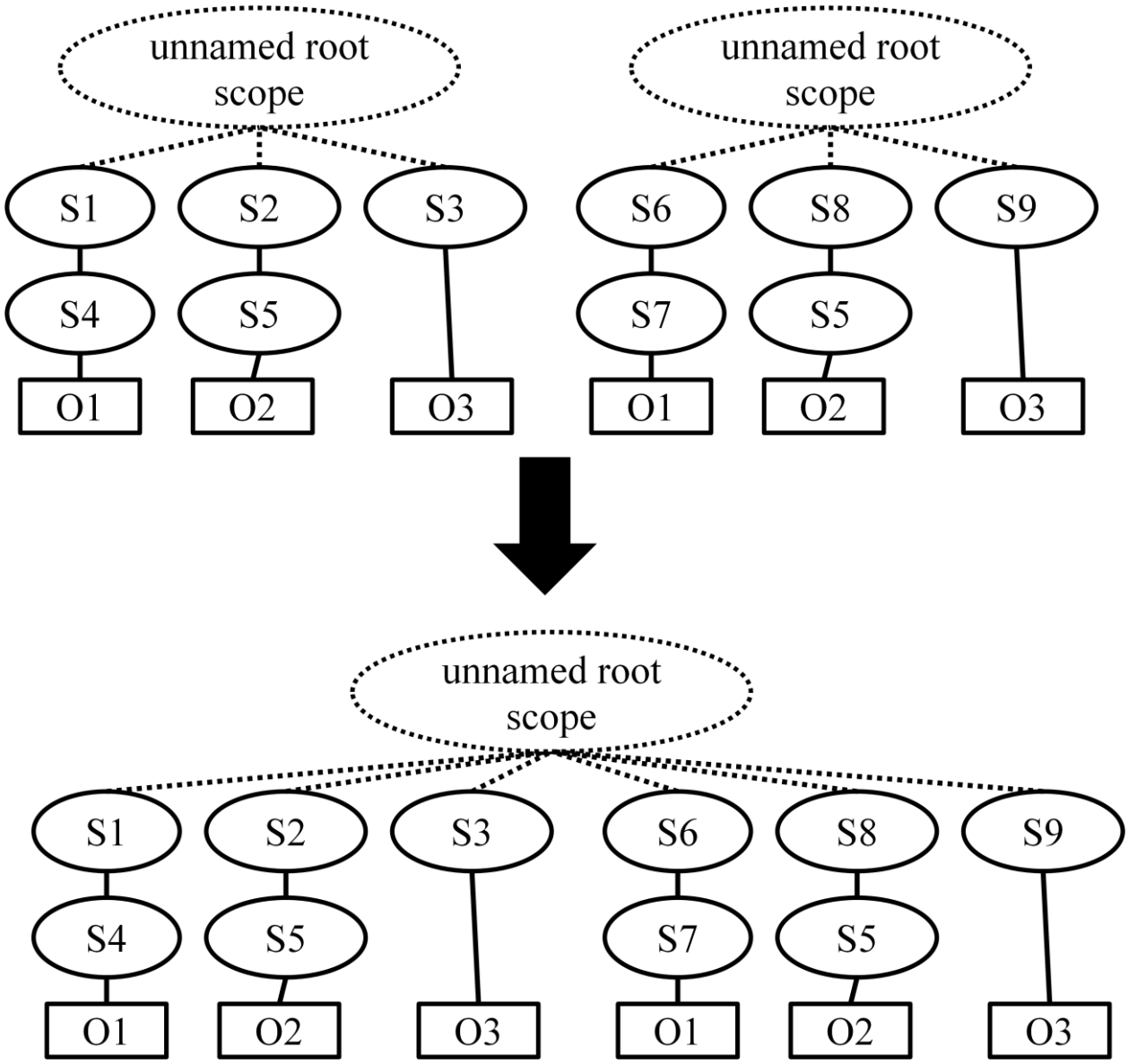


Figure 16. Joining two scopes in a simple case. The top half depicts the two independent scopes before joining, while the bottom half depicts the scope after joining.

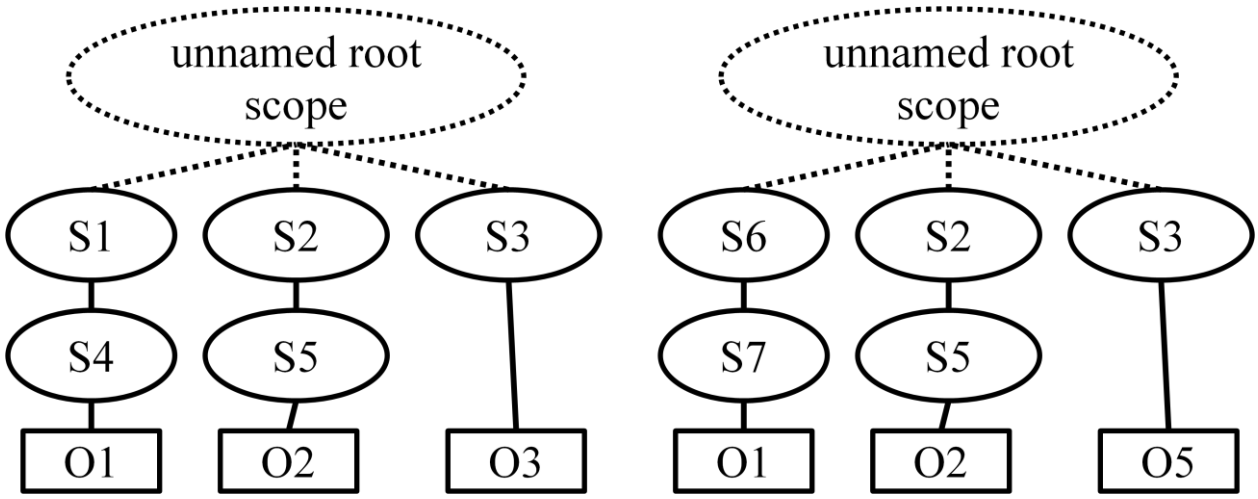


Figure 17. Two scopes with conflicting paths

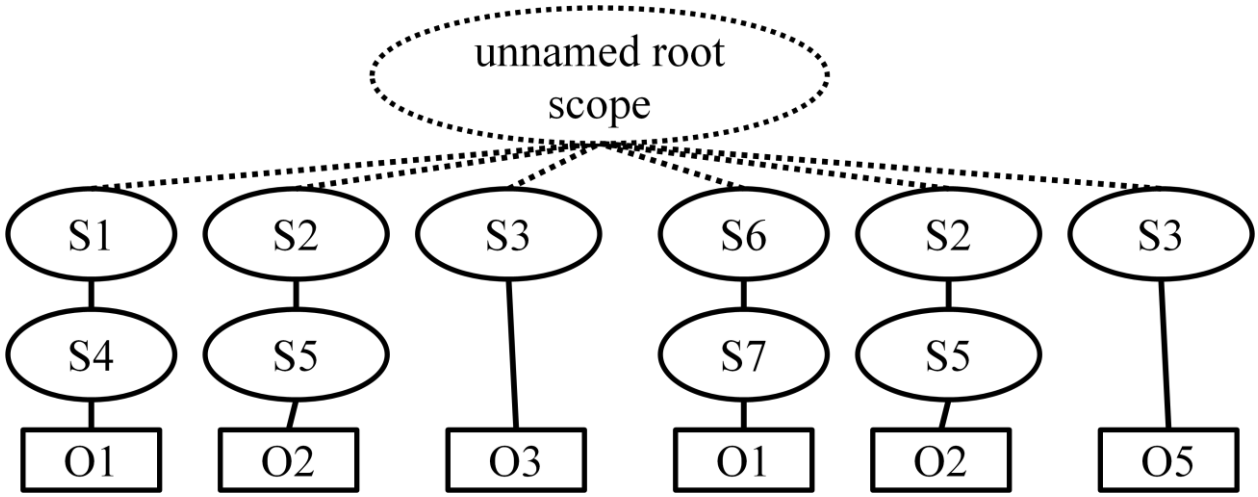


Figure 18. Joining two scopes with conflicting paths using the simple method

Our solution was to name the unnamed root scopes of the independent scopes to encapsulate those scopes into their own worlds (Figure 19). As you can see from the figure, we named the previously-unnamed root scopes $S8$ and $S9$. This solved the path conflict because the full path to $O2$ in the left scope now became $/S8/S2/S5/O2$, and the full path to $O2$ in the right scope now became $/S9/S2/S5/O2$.

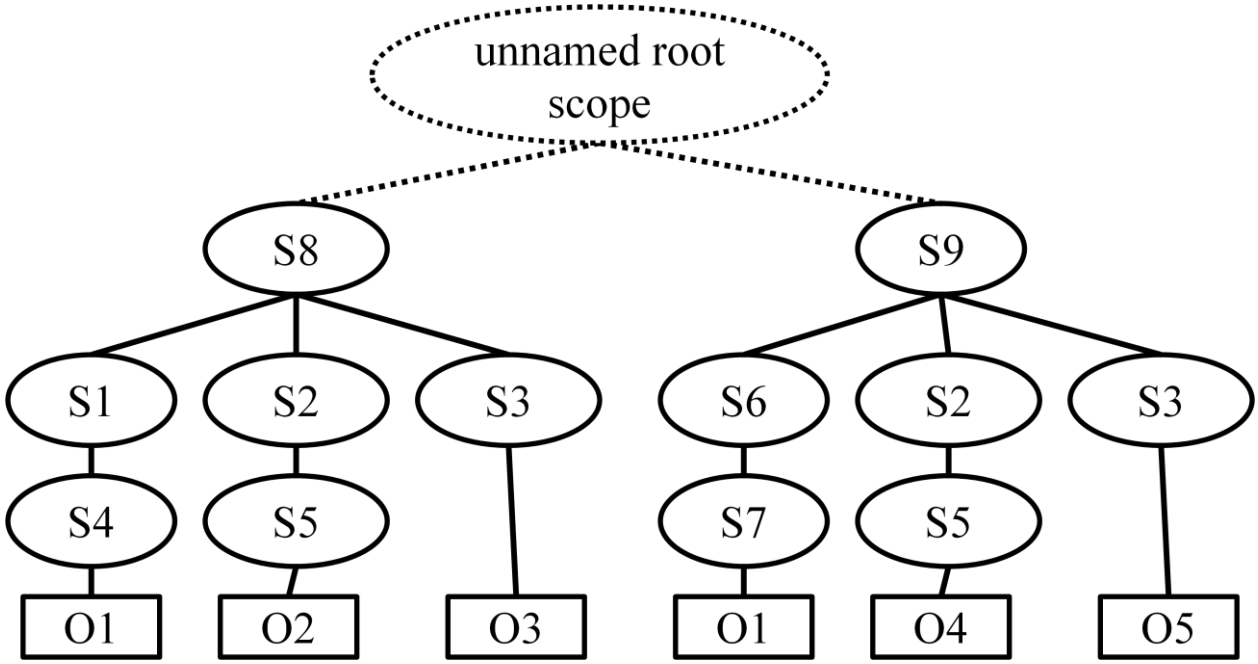


Figure 19. Joining two scopes with conflicting paths by creating two new scopes

In addition to resolving path conflicts, creating new scopes also has a second benefit which helps organization. When two scopes are created separately, it is should be organized differently. Going back to the example in Figure 14, notice that the $East\ Coast$ scope and the $US\ Cities$ scope are both part of the USA scope. It is possible that these two scopes were once unnamed root scopes independent of each other. When the need for a centralizing USA scope appeared, the scopes $East\ Coast$ and $US\ Cities$ were created to allow not only for conflict-free scope-joining but also for organizational purposes. It makes sense to keep the information in those scopes separate because the information in them are already grouped in their own way.

6.2.2 Impact of Scope-joining on Scaling

Scope-joining has both positive and negative effects on scaling in PSIRP. It is a good feature to have because it allows scopes to join together to form a larger scope. That means there are no limitations to the number of independent scopes. We can have a large amount of scopes without worrying about other scopes and conflicting identifiers. This allows for scaling in terms of adding more independent scopes. Then, whenever we find commonality between some scopes and wish to join them together, we could just name the previously-unnamed root scopes and put them in a new unnamed root scope. This makes scaling easier since it is a way to easily add and create scopes to a system. The idea of a global root scope across all scopes does not exist, and one is able to always add on root scopes.

However, continuously adding scopes also reveals a problem. Scope-joining adds an complexity to the scope structure. As more scopes join together, the scope structure as a whole becomes deeper and wider. The larger the scope structure becomes, the harder it is to search for a specific scope. Compounded with Chord's failure to scale for an information-sharing network management situation, it will be a challenge to scale with scope-joining.

6.3 Traversing Through PSIRP Nodes

A useful thing to have for extracting information from a tree-like structure is the ability to traverse the tree in order. This improves scalability because a node only needs to know a relative path to an object instead of absolute path that will force it to find the location of the object. Thus, we decided to inspect PSIRP's capabilities in traversing nodes in its tree. Our results showed that PSIRP does not support this very well.

6.3.1 Example of Traversing through PSIRP Nodes

To find PSIRP's tree-traversal behavior, we started off by describing an example case. There was a root scope which we called *Packet traces* for readability. It contained two sub-scopes called *Incoming Ports* and *Ports*. *Incoming Ports* was a scope that had all the packet traces for the incoming port. *Ports* was a scope that contained all packet traces on both incoming ports and outgoing ports. See Figure 20 for reference. The gray rectangles in the figure represent the information objects corresponding to packet traces from incoming ports only, while the white rectangles represent information objects containing packet traces for outgoing ports only. The two lines in the information objects represent different naming schemes for each of the intermediate scopes. The top row are the identifiers *Ports* uses to describe the packets, the bolded numbers in the bottom row are identifiers *Incoming Ports* uses to name its information, and the italicized numbers in the second row are the names *Outgoing Ports* gives to those white information objects.

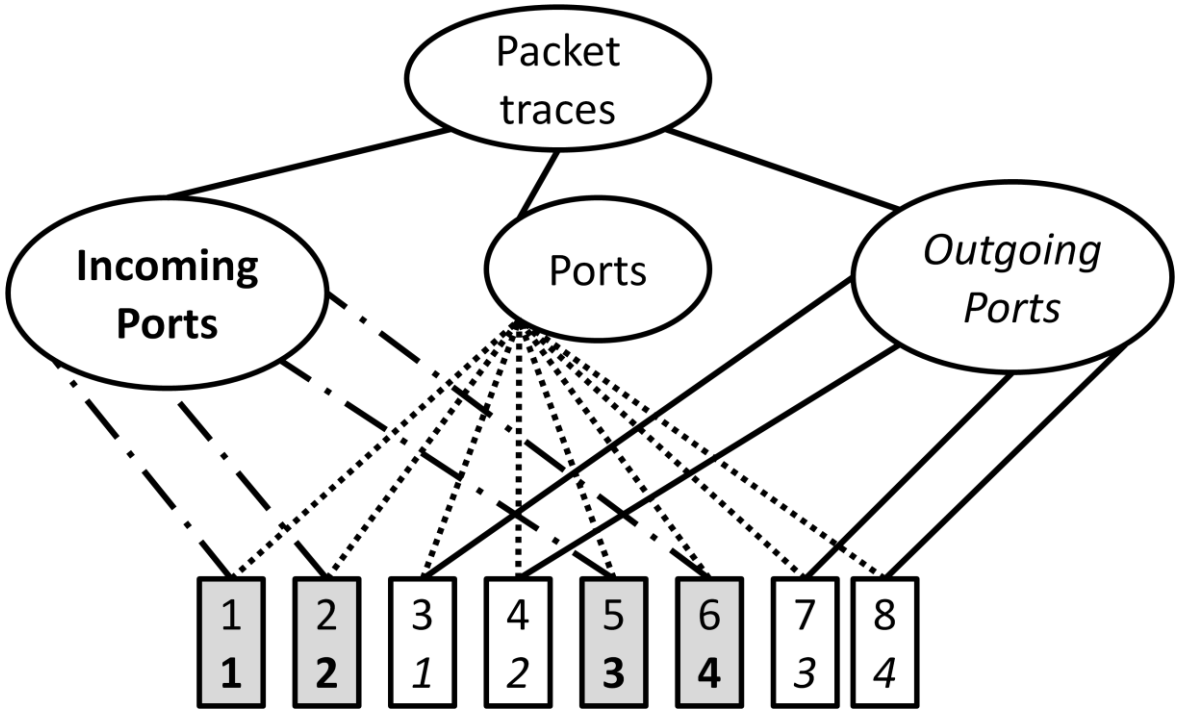


Figure 20. Simple PSIRP System for packet traces

For PSIRP to traverse the tree, we assumed the identifiers were path names that allowed us to move down the tree. The method of traversal was Depth First. Our goal was to use some function *getNextTrace* that would traverse depth first through the subsequent scopes and return the traces for us. The results are shown in Figure 21.

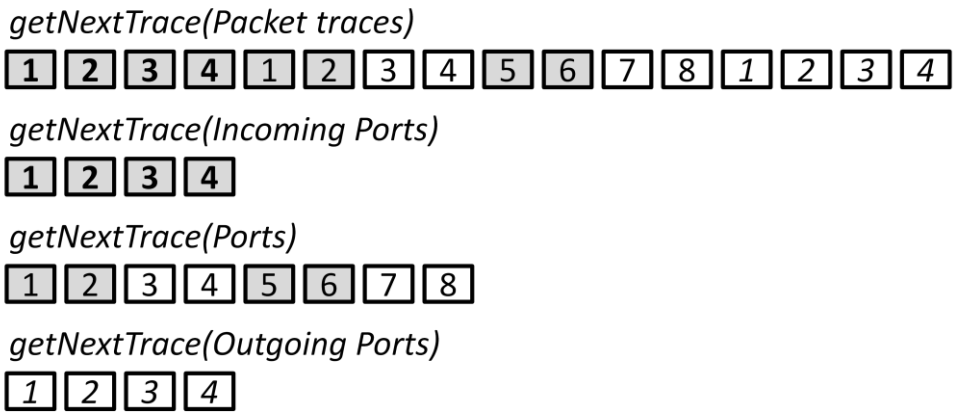


Figure 21. Returns from getNextTrace

6.3.2 Analysis of Results from Traversing through PSIRP Nodes

The first thing to notice immediately is the different types of packet traces that $getNextTrace$ can return – namely, the incoming packet traces and outgoing packet traces. The results show that the returns vary greatly depending on the scope. $getNextTrace(Incoming\ Ports)$ only returned incoming packet traces, while $getNextTrace(Outgoing\ Ports)$ only returned outgoing packet traces. The expected results would have likely been $getNextTrace(Ports)$. What's even more interesting is that because information can be in many different scopes, we ended up getting the same packet traces twice using $getNextTrace(Packet\ traces)$. Thus, we can see that $getNextTrace$ does not always give us expected results and instead is even confusing to use because of its inconsistent functionality.

The ambiguity of $getNextTrace$'s output led us to believe that PSIRP's naming convention does not allow easy traversals. In turn, this means PSIRP will not be able to effectively use relative paths. As a result, PSIRP's best way of extraction is still by searching for full path names of objects. This is not good for scalability because it requires scopes to know more about the system to find an object instead of finding a path from itself to that object. As the system scales up, there will be more that needs to be known in the scopes.

6.4 Conclusion of Scaling in PSIRP

After looking at a few aspects of PSIRP that could help or deter scaling, we found that they all do not allow PSIRP to scale well. One major reason behind a lot of the findings was the fact that information objects could exist in more than one scope. This changed the behavior of common events such as tree searching and Depth First traversals, giving unexpected results that hurt scalability.

7 Conclusion

7.1 Overview

The numerous network management domains and the need to share information across the domains has led to a desire for a new type of network architecture that finds and retrieves a needle in a haystack quickly. This notion of centering a network around information is not a new one. In fact, current generic usage of the network has also led to an interest in information-sharing. Thus, we decided to look at an information-centric network architecture, PSIRP, and apply it to the area of network management. Information-centric world suits network management perfectly because it involves collecting large amounts of information daily and looking through the information when problems occur across the network. Because of the continuous wealth of data influx, it is crucial to find a method that not only lets us share information quickly but also scales well.

Scaling plays a large role in deciding whether or not an information-centric system should be used. If a network does not scale well, it will face more and more difficulties getting the correction information across in a timely manner. There are two parts to scaling in network management. The first part is quantity. In order to scale well, the system must be able to handle a large amount of quantity. The second part is distribution. As the network grows, information spreads out to more areas from a growing number of different devices. In order to scale well, the system needs to also account for the outward expansion of information. Thus, we set out to investigate the scaling in PSIRP. Because PSIRP is closely tied with Chord, we also decided to find a solution to make Chord scale better.

7.2 Discussion of Our Analysis Findings

The way we expanded Chord to handle massive amounts of data was to try having nested Chord systems. We experimented to see whether more nested Chord systems would have a positive or negative impact on the system. Our findings showed that the best solution would be to have one server in each Chord system and to have the maximum amount of nests possible for the number of data required. However, we decided this approach was impractical because it does not inherit the advantages that a regular Chord system. Scaling both in terms of quantity and distribution contradict in the single-server case. Thus, there needs to be a balance between the number of Chord layers and the number of servers in a Chord system and a tradeoff between performance and distribution.

Our conclusions from analyzing Chord merely told us the physical structure in which PSIRP should organize its scopes for quick lookup. However, the Chord analysis did not look at other parts of PSIRP that could impact its scalability. In particular, we focused on the logical structure of the PSIRP scopes, the behavior of scope-joining, and the traversal through PSIRP nodes.

For the first point, our results showed that PSIRP scope trees grow from bottom up, which differs from BGP's AS tree. BGP's AS tree scales because there are always only a few root nodes at the top of the tree, meaning one needs to only look through a small search space when trying to find something at the top level. The difference in PSIRP scopes means that PSIRP will need to search through a large number of scopes the higher up it goes in the tree. Again, this points to the need for some mechanism that provides fast lookup while handling a large number of entries. Because our Chord analysis did not show a clear way for Chord to scale well, the lookups at higher levels of the tree may be slow and the designer will need to make tradeoffs.

For the second point, our results showed that the lack of a fixed global root scope means that scopes can get larger and larger. The larger the scopes get, the deeper the scope tree gets. As shown in the first point, it is challenging for the scope tree to grow because lookup times become slower.

For the third point, our results showed that calling the same functions on different scopes may result in different data than the ones expected. The problem with this is that it makes it difficult to refer to scopes using relative paths. As a result, as the system grows, names and relative paths will become more confusing and inconsistent. This may lead to retrieving incorrect data.

Looking at all our findings, it seems that there are some challenges in scaling that arise from PSIRP scopes. That said, we also note that the advantage of PSIRP's Chord-based approached is that the designer is given an $O(\log^2 N)$ performance and distribution tradeoff.

7.3 Future Work

There is still much to be done in the space of finding scalable information-sharing for network management. The next step could be to implement prototypes of the different information-centric architectures out there (PSIRP, CCN, DONA, NetInf) and run tests on them to get specific numbers on how they scale and how they compare to each other. Another possible path to take is to take a deeper look into PSIRP and think of ways to improve its design so that it solves the specific problems we've mentioned. Regardless of which future steps are taken, it is important to keep looking forward and investigating ways to make an information-centric network scale with respect to network management.

8. References

[1] Ahlgren, B., M. D'Ambrosio, C. Dannewitz, A. Eriksson, et al. *Second NetInf Architecture Description*. Tech. no. D-6.2. 2010.

[2] Balakrishnan, H., and N. Feamster. "Interdomain Internet Routing." Lecture. 2005.

[3] Eugster, P. Th., P. A. Felber, R. Guerraoui, and A. Kermarrec. "The Many Faces of Publish/subscribe." *ACM Computing Surveys* 35.2 (2003): 114-31.

[4] Ganesan, P., K. Gummadi, and H. Garcia-Molina. *Canon in G Major: Designing DHTs with Hierarchical Structure*. Proc. of ICDCS '04 Proceedings of the 24th International Conference on Distributed Computing Systems. IEEE, 2004. 263-72.

[5] Jacobson, V. "A New Way to Look at Networking." Google Tech Talk. 30 Aug. 2006. Speech.

[6] Jacobson, V., D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. BRaynard. *Networking Named Content*. Proc. of CoNext '09, Rome. ACM, 2009.

[7] Koponen, T. *A Data-Oriented Network Architecture*. Diss. Helsinki University of Technology, 2008.

[8] Koponen, T., M. Chawla, B. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica. "A Data-Oriented (and Beyond) Network Architecture." *ACM SIGCOMM Computer Communication Review* 37.4 (2007): 181.

[9] Moy, J. *RFC 1583: OSPF Version 2*, March 1994

[10] Rajahalme, J., M. Särelä, K. Visala, and J. Riihijärvi. *Inter-Domain Rendezvous Service Architecture*. Tech. no. TR09-003. 2009.

[11] Sollins, K. *An Architecture for Network Management*. Proc. of ReArch '09. ACM, 2009.

[12] Stoica, I., R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications." *ACM SIGCOMM Computer Communication Review* 31.4 (2001): 149-60.

[13] Trossen, D., P. Nikander, K. Visala, T. Burbridge, P. Botham, et al. *Publish-Subscribe Internet Routing Paradigm*. Tech. no. D2.5. 2009.