The Design and Implementation of Declarative Networks

by

Boon Thau Loo

B.S. (University of California, Berkeley) 1999
M.S. (Stanford University) 2000

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Joseph M. Hellerstein, Chair
Professor Ion Stoica
Professor John Chuang

Fall 2006

The dissertation of Boon Thau Loo is approved:

Professor Joseph M. Hellerstein, Chair	Date
Professor Ion Stoica	Date
Professor John Chuang	Date

University of California, Berkeley

Fall 2006

Abstract

The Design and Implementation of Declarative Networks

by

Boon Thau Loo

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Joseph M. Hellerstein, Chair

In this dissertation, we present the design and implementation of *declarative networks*. *Declarative networking* proposes the use of a declarative query language for specifying and implementing network protocols, and employs a dataflow framework at runtime for communication and maintenance of network state. The primary goal of declarative networking is to greatly simplify the process of specifying, implementing, deploying and evolving a network design. In addition, declarative networking serves as an important step towards an extensible, evolvable network architecture that can support *flexible*, *secure* and *efficient* deployment of new network protocols.

Our main contributions are as follows. First, we formally define the *Network Datalog* (*NDlog*) language based on extensions to the Datalog recursive query language, and propose *NDlog* as a Domain Specific Language for programming network protocols. We demonstrate that *NDlog* can be used to express a large variety of network protocols in a handful of lines of program code, resulting in orders of magnitude reduction in code size. For example, the Chord overlay network can be specified in 48 *NDlog* rules. In addition, the core of the language (Datalog) has polynomial complexity, and our *NDlog* extensions can be statically analyzed for termination using standard analysis techniques.

Second, to validate the design of *NDlog*, we present our implementation of P2, which is a full-fledged declarative networking system that compiles *NDlog* and executes it via a dataflow engine inspired by the Click modular router. We experimentally evaluate the P2 system on hundreds of distributed machines. The P2 system is publicly available for download and has been used in research projects at a number of institutions.

Third, based on our experiences implementing declarative networks, we explore a wide variety of database research issues that are important for the practical realization of declarative networks. These include pipelined execution of distributed recursive queries, reasoning about query semantics based on the well-known distributed systems notion of "eventual consistency", incorporating the notion of soft-state into the logical framework of *NDlog*, and the use of query optimizations to improve the performance of network protocols.

Professor Joseph M. Hellerstein, Chair Date

Acknowledgements

I would like to thank my outstanding co-advisors Joe Hellerstein and Ion Stoica. Joe and Ion have been the perfect combination of advisors, and I feel very lucky to have the opportunity to work closely with both of them.

Joe Hellerstein has been a role model and an incredible advisor throughout my years at Berkeley. I met Joe eight years ago through Honesty Young while I was an undergraduate summer intern at IBM Almaden Research Center. After the summer, Joe generously took me under his wings and gave me the opportunity to work on research projects at Berkeley. Joe is truly a visionary of databases and their applications to networking. Since I came back to Berkeley as a doctoral student, he has infected me and several others with his enthusiasm on new research directions at the boundaries of databases and networking. Joe has been amazingly generous with his time and attention. Apart from his advice on technical issues, he has helped me in many other areas, including writing, giving presentations, working with people, balancing family and work, strategies for job search, and the list goes on. I can never thank him enough for the positive impact that he has on my life.

In the last five years, I have also been very fortunate to work with Ion Stoica. Ion combines depth of his technical knowledge with big picture vision. He has taught me how to be a better researcher, through his focus, discipline, strive for perfection, and combining attention to details with understanding of the big picture. My dissertation and graduate career would not have been the same without Ion. He introduced me to the exciting world of networking research, connected me with members of the networking community and helped position my work to be of value to them.

During the course of this dissertation, I am grateful to Raghu Ramakrishnan from the University of Wisconsin-Madison who spent substantial time helping me with my research even though I am not his official student. His deep knowledge of deductive databases was instrumental in helping me focus on relevant literature, and more importantly, identify new database research challenges that are essential for the practical realization of declarative networks. I leave graduate school with fond memories of the time we spent at UW-Madison and UC Berkeley brainstorming on recursive query processing.

While working on the PIER project, Scott Shenker offered me many insights into the relevance of database technologies to networked systems. Mike Franklin has given me great advice on doing research, giving presentations, speaking up in class, and strategies for job search. Together with John Chuang, Mike attended my qualifying examination and gave me valuable feedback on this work. I also like to thank John Chuang for participating on my dissertation committee.

My research in graduate school centered around the PIER and P2 projects. In the context of these two projects, I have had the privilege of collaborating extensively with two outstanding graduate students from the database group: Ryan Huebsch on the PIER project, and Tyson Condie on the P2 project. Both of them made significant contributions to the codebases that made my research possible. In addition, I would also like to thank the rest of the P2 team which includes Minos Garofalakis, David Gay, Tristan Koo, Petros Maniatis, and Timothy Roscoe from Intel Research Berkeley (IRB), Atul Singh from Rice University, and Alex Rasmussen from UC Berkeley. The P2 system would not have been realized without the collective efforts of everyone. I am proud to be a part of this wonderful team.

The NMS and PDOS groups at MIT graciously hosted Joe and myself during the second year of my Ph.D. Apart from writing one of my first papers with Jinyang Li et al., several ideas presented in this dissertation trace their roots back to that period. I am also grateful to the researchers at Microsoft Research Silicon Valley for hosting me during the final stages of this dissertation. In addition to generating some of the new ideas in the future work section, significant improvements were made to this dissertation during my stay there.

My appreciation also goes to the other graduate students from the database group for their feedback on my work, stimulating discussions and great companionship. They include Sirish Chandrasekaran, David Chu, Owen Cooper, Matt Denny, Amol Deshpande, Yanlei Diao, Shawn Jeffery, Sailesh Krishnamurthy, David Liu, Sam Madden, Alexandra Meliou, Fred Reiss, Shariq Rizvi, and Mehul Shah. Several graduate students from the networking and systems groups provided valuable feedback on my papers and talks, especially during the early stages of this work. They include Cheng Tien Ee, Byung-Gon Chun, Ling Huang, Karthik Lakshminarayanan, Sylvia Ratnasamy, Sean Rhea, Lakshmi Subramanian, Fang Yu, and Shelley Zhuang. Nina Taft from IRB gave thoughtful and detailed feedback for my first SIGCOMM submission on declarative routing. Outside of research, Rahul Aggrawal, Ling Huang, Matthew Kam, Lynn Nyane, Godfrey Tan, Weehong Tan, and several members of the UC Berkeley Singapore Malaysia Students' Association have made my ten years in America so much easier.

The Monetary Authority of Singapore and the Kent Ridge Digital Labs in Singapore sponsored my undergraduate and masters degrees. Their generosity allowed me to fulfill my dream of studying computer science in the U.S. and to see the world. My life would have been very different without the scholarships. Limsoon Wong and Juzar Motiwalla have been influential in my decision to embark on a career in research.

My family in Singapore has been very supportive of me all these years. Even though they live halfway around the world, my parents, sister, and brother have been constant sources of encouragement. They cheer for me in good times, and comfort me in bad times. In particular, my mum has been my biggest source of support. Despite her own difficulties with kidney dialysis three times a week, she has never failed to stay positive and strong. In times of difficulty in America, I always draw inspiration from her positive outlook in life. I am also grateful to the National Kidney Foundation of Singapore for their support of my family in the past ten years.

I am deeply indebted to my wife Shiyan for her love and understanding. In the past twelve years since we met, almost half the time was spent in graduate school. While the route to a Ph.D. has not been an easy one for me, it is even harder for my immediate loved ones. Shiyan has made tremendous sacrifices so that I could focus on my work. The biggest joy of our lives, Kwanchi was born while I was finishing this dissertation. Shiyan, I thank you for everything and dedicate this dissertation to you. I look forward to continuing our journey together with Kwanchi.

Dedicated to my wife Shiyan

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The Internet faces many challenges today, ranging from the lack of protection against unwanted or harmful traffic to the increasing complexity and fragility of inter-domain routing. In addition, the proliferation of new applications on the Internet has also led to growing demands for new functionalities such as mobility, content-based routing, and quality-of-service (QoS) routing. As a result, there is an increasing consensus in the networking community that the current Internet architecture is fundamentally ill-equipped to handle the needs of future uses and challenges [8].

A radical approach suggested recently calls for a "clean-slate" redesign of the Internet, by revisiting its original design goals and principles [31]. These clean-slate proposals have received substantial attention from the research community, as reflected by major NSF initiatives such as Future INternet Design (FIND) [46] and Global Environment for Network Innovations (GENI) [48].

At the same time, given the existing limitations of the Internet, the evolutionary approach that has been widely adopted today involves the deployment of overlay networks [93] on the Internet. An overlay network is a virtual network of nodes and logical links that is built on top of an existing network with the purpose of implementing a network service that is not available in the existing network. Examples of overlay networks on today's Internet include commercial content distribution networks such as Akamai [7], peer-to-peer (P2P) applications for file-sharing [50] and telephony [113], as well as a wide range of experimental prototypes running on the PlanetLab [94] global testbed.

While overlay networks have been successfully used to support a variety of distributed applications, they are viewed by some as an incremental stop-gap solution. Whether one is a proponent of the revolutionary clean-slate redesign or the evolutionary approach of using overlay networks, it is clear that we are entering a period of significant flux in network services, protocols and architecture.

1.2 Main Contributions

In this dissertation, we present the design and implementation of *declarative networks*. *Declarative networking* proposes the use of declarative query language for specifying and implementing network protocols, and employs a dataflow framework at runtime for communication and the maintenance of network state. The key idea behind declarative networking is that declarative recursive queries [6; 98], which are used in the database community for querying graph structures, are a natural fit for expressing the properties of various network protocols.

The primary goal of declarative networking is to greatly simplify the process of specifying, implementing, deploying and evolving a network design. In addition, declarative networking serves as an important step towards an extensible, evolvable network architecture that can support *flexible*, *secure* and *efficient* deployment of new network protocols. Existing solutions are either highly efficient but inflexible (*e.g.*, hard-coded network protocols today), or highly flexible but insecure (*e.g.*, *active networks* [119]). By using a declarative language that is more amenable to static analysis than traditional programming languages, we are able to strike a better balance between flexibility and security than existing solutions.

To realize the vision of declarative networking, this dissertation makes the following key contributions:

- First, we formally define the **Network Datalog (*NDlog*)** language based on extensions to Datalog, a traditional recursive query language used in the database community. *NDlog* builds upon traditional Datalog to enable *distributed* and *soft-state* computations with *restricted communication* based on the underlying physical connectivity, all of which are essential in the network setting.

- In **declarative routing** [79; 80], we demonstrate the use of the declarative framework for building an extensible routing infrastructure that provides flexibility, yet retains in large the efficiency and security of today's routing protocols. We show that *NDlog* programs are a natural and compact way of expressing a variety of well-known routing protocols, typically in a handful of lines of program code. This allows ease of customization, where higher-level routing concepts (*e.g.*, QoS constraints) can be achieved via simple modifications to the *NDlog* programs. We also show these these programs can be statically analyzed for termination using standard database techniques [67]. and are amenable to well-studied query optimizations from the database literature.

- In **declarative overlays** [76], we show that *NDlog* can be used to implement complex application-level overlay networks such as multicast overlays and distributed hash tables (DHTs). We demonstrate a working implementation of the Chord [114] overlay network specified in 48 *NDlog* rules, versus thousands of lines of C++ for the original version, resulting in orders of magnitude reduction in code size.

- To validate the design of *NDlog*, we present our implementation of P2 [2], which is a full-fledged declarative networking system with a dataflow engine inspired by the Click modular router [65]. The P2 system takes as input *NDlog* programs, compiles them into distributed execution plans that are then executed by a distributed dataflow engine to implement the network protocols. We experimentally evaluate the P2 system on hundreds of distributed machines running a wide variety of different protocols, including traditional routing protocols as well as complex overlays such as distributed hash tables. The P2 system is publicly available for download and has been used in research projects at various institutions.

1.3 Distributed Recursive Query Processing

Recursive query research has often been criticized as being only of theoretical interest and detached from practical realities [21]. In this dissertation, we not only demonstrate the practical use of recursive queries outside of the traditional domain of stored centralized databases, we also identify and address several important and challenging database research issues [75] that are essential for the practical realization of declarative networking. Specifically, the recursive query processing issues that we tackle in this dissertation include the following:

- First, we extend existing techniques for recursive query processing to a distributed context in order to generate distributed dataflow-based execution plans for *NDlog* programs.

- Second, based on the execution model of our distributed dataflows, we introduce and prove correct relaxed versions of the traditional, centralized execution strategy known as *semi-naïve* [11; 12; 15] fixpoint evaluation. Our techniques, called *buffered semi-naïve* and *pipelined semi-naïve* evaluation, overcome fundamental problems of semi-naïve evaluation in an asynchronous distributed setting, and should be of independent interest outside the context of declarative networking: they significantly increase the flexibility of semi-naïve evaluation to reorder the derivation of facts.

- Third, in the network setting, transactional isolation of updates from concurrent queries is often inappropriate; network protocols must incorporate concurrent updates about the state of the network while they run. We address this by formalizing the typical distributed systems notion of "eventual consistency" in our context of derived data. Using techniques from materialized recursive view maintenance, we incorporate updates to input tables during *NDlog* program execution, and still ensure well-defined eventual consistency semantics. This is of independent interest beyond the network setting when handling updates and long-running recursive queries.

- We cleanly incorporate the notion of soft state[1] into the logical framework of *NDlog*, describe new query processing and view maintenance techniques to process soft-state data, and demonstrate how the distributed systems notion of "eventual consistency" can be similarly achieved as above.

- We survey a number of automatic optimization opportunities that arise in the declarative networking context, including applications of traditional database techniques such as aggregate selections [118; 47] and magic-sets rewriting [16; 18], as well as new optimizations we develop for work-sharing, caching, and cost-based optimizations based on graph statistics. Many of these ideas can be applied to query processing engines outside the context of declarative networking or distributed implementations.

[1]Network state is typically maintained as soft state [31; 99] for reasons of robustness and scalability.

While recursive query processing is considered a mature field, in the course of this work, we raise new interesting database research challenges motivated in the distributed setting that we hope can open new avenues for research in the theory, implementation and application of recursive queries.

1.4 Overview of Declarative Networks

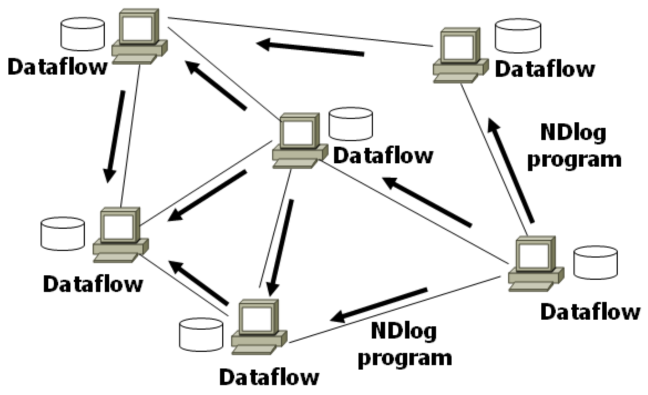

Figure 1.1: A Declarative Network

Figure 1.1 illustrates a declarative network at a conceptual level. Like any traditional network, a declarative network maintains network state at each node to enable the routing and forwarding of packets. The network state is stored as relational tables distributed across the network, similar to a traditional distributed database [90]. Network protocols are declaratively specified as distributed recursive queries over the network state. Recursive queries have traditionally been used in the database community for posing queries over graph structures in deductive databases. The main observation that inspired this work on declarative networking is that these recursive queries are a natural fit for expressing network protocols, which themselves are based on recursive relations among nodes in the network.

The recursive query language that we develop is a distributed variant of Datalog called *Network Datalog* (*NDlog*). Intuitively, one can view the forwarding tables generated by network protocols as the output of distributed recursive queries over changing input network state (network links, nodes, load, operator policies, etc.), and the query results need to be kept consistent at all times with the changing network state.

Network protocols are specified as *NDlog* programs and disseminated in the network. Upon receiving *NDlog* programs, each node compiles the declarative specifications into execution plans in the form of distributed dataflows. When executed, these dataflows generate message exchanges among nodes as well as network state modifications, resulting in the implementation of the network protocols. Multiple declarative networks can run simultaneously, either as separate dataflows, or compiled into a single dataflow where common functionalities among the protocols can be shared.

1.5 The Case for Declarative Networking

Declarative networking presents three advantages over existing approaches: *ease of programming*, *optimizability* and *balance between extensibility and security*. We summarize the advantages in the rest of this section. This dissertation focuses on the first two advantages: ease of programming and optimizability. In addition, program analysis techniques exist today that take advantage of the formal, high-level nature of a declarative language, as we discuss in Section 1.5.3. There are opportunities to extend these analysis techniques that are outside the scope of the dissertation. We return to this topic as future work in Chapter 10.

1.5.1 Ease of Programming

A declarative language allows us to specify at a high level "what" to do, rather than "how" to do it. When feasible, the declarative approach can lead to ease of programming and significant reduction in code size. As we demonstrate in Chapter 3, *NDlog* can express a variety of well-known routing protocols (*e.g.*, distance vector, path vector, dynamic source routing, link state, multicast) in a compact and clean fashion, typically in a handful of lines of program code. Moreover, higher-level routing concepts (*e.g.*, QoS constraints) can be achieved via simple modifications to these programs. Furthermore, in Chapter 6, we show that complex application-level overlay networks can also be expressed naturally in NDlog.

Declarative network descriptions can be extremely concise. For example, the Chord overlay network can be specified in 48 *NDlog* rules, versus thousands of lines of code for the MIT Chord reference implementation, and more than 320 statements for a less complete implementation in the Macedon [105] domain-specific language for overlays (see related work in Chapter 9 for comparisons with Macedon). Also, the high-level, declarative nature of P2 specifications means that they decompose cleanly into logically reusable units: for example, a Symphony DHT [83] might share many of the definitions in the Chord specification. Moreover, by providing a uniform declarative language for distributed querying and networking, we enable the natural integration of distributed information-gathering tasks like resource discovery and network status monitoring.

In addition to ease of programming, there are other advantages to the use of a high level language. For example, *NDlog* specifications can illustrate surprising relations between network protocols, as we illustrate in Chapter 8. In particular, we show that path vector and dynamic source routing protocols differ only in a simple, traditional database optimization decision: the order in which a query's predicates are evaluated. The use of higher-level abstractions also provides the potential to statically check network protocols for security and correctness properties [43]. Dynamic runtime checks to test distributed properties of the network can also be easily expressed as declarative queries, providing a uniform framework for network specification, monitoring and debugging [112].

1.5.2 Optimizability

Declarative networking achieves performance comparable to traditional approaches. Moreover, by using a declarative framework rooted in databases, we can achieve even better performance by utilizing query processing and optimization techniques that are well-studied in the database community.

Our declarative approach to protocol specification reveals new opportunities for optimizing network protocols. First, the use of a high-level declarative language facilitates the identification and sharing of common functionalities among different declarative networks. Second, off-the-shelf database optimization techniques can be applied to declarative routing specifications to achieve tangible performance benefits. Third, we develop new optimization techniques suited to the distributed, soft-state context of network protocols.

1.5.3 Balance of Extensibility and Security

In addition to the benefits of having a higher-level, compact specification, declarative networking achieves a better balance between *extensibility* and *security* compared to existing solutions. Extensibility, or the ability to easily add new functionality to existing systems, is an important requirement in our setting as a means of rapid deployment and experimentation with network protocols. However, extensibility has traditionally been achieved at the expense of security [115; 22]. In the network domain, this concern is best illustrated by active networks [119] which, at the extreme, allow routers to download and execute arbitrary code. While active networks provide full generality, security concerns have limited their practical usage.

Declarative networking can be viewed as a safer, restricted instantiation of active networks, where our approach essentially proposes *NDlog* as a Domain Specific Language (DSL) for programming a network. The core of *NDlog* is Datalog, which has complexity polynomial in the size of the network state [6]. While our language extensions to *NDlog* alter its theoretical worst-case complexity, there exist static analysis tests on termination for a large class of recursive queries [67]. This addresses the *safety* aspect of security, where specified protocols can now be checked to ensure that they do not consume infinite resources before execution. In addition, by "sandboxing" *NDlog* programs within a database query engine, we are also able to contain undesirable side-effects during query execution.

1.6 Organization

This dissertation is organized as follows. In Chapter 2, we provide an introduction to Datalog, and then motivate and formally define the *NDlog* language. Given the *NDlog* language, we demonstrate in Chapter 3 the expressiveness of *NDlog* in compactly specifying declarative routing protocols that implement a variety of well-known routing protocols.

We next demonstrate how declarative routing protocols can be realized in practice by describing the implementation of the P2 declarative networking system in Chapter 4, and query processing techniques for compiling *NDlog* programs into P2 execution plans in Chapter 5.

In Chapter 6, we further apply the declarative framework to more challenging scenarios, where we use *NDlog* to specify complex overlay networks such as the Narada mesh [30] for end-system multicast and the Chord distributed hash table. Our Chord implementation is roughly two orders of magnitude less code than the original C++ implementation.

To validate declarative networking concepts, in Chapter 7, we present evaluation results from a distributed deployment on the Emulab [40] network testbed, running prototypes of declarative networks. In Chapter 8, we discuss and evaluate a number of query optimizations that arise in the declarative networking context.

We present in Chapter 9 a survey of related work in both the database and networking domains. We then conclude in Chapter 10 summarizing the overall impact of declarative networking in the past few months, a discussion of open issues and future research directions.

Chapter 2

The Network Datalog Language

In this chapter, we formally define the Network Datalog (*NDlog*) language for declarative networking. The *NDlog* language is based on extensions to traditional Datalog, a well-known recursive query language traditionally designed for querying graph-structured data in a centralized database.

The chapter is organized as follows. In Section 2.1, we provide an introduction to Datalog. In Section 2.2, we present the *NDlog* language using an example program that computes all-pairs shortest paths in a network from specification to execution. Based on this example program, we highlight the *NDlog* extensions to traditional Datalog, and show that the execution of this program resembles a well-known routing protocol for computing shortest paths in a network.

Following the example, in Sections 2.3, 2.4, 2.5 and 2.6, we formally describe the data and query model of *NDlog* that addresses its four main requirements: *distributed computation*, *link-restricted communication*, *soft-state data and rules*, and *incremental maintenance of network state*.

2.1 Introduction to Datalog

We first provide a short review of Datalog, following the conventions in Ramakrishnan and Ullman's survey [98]. A Datalog program consists of a set of declarative *rules* and an optional *query*. Since these programs are commonly called *"recursive queries"* in the database literature, we use the term "query" and "program" interchangeably when we refer to a Datalog program.

A Datalog *rule* has the form *p :-* $q_1, q_2, ..., q_n$., which can be read informally as "q_1 and q_2 and ... and q_n implies p". p is the *head* of the rule, and $q_1, q_2, ..., q_n$ is a list of *literals* that constitutes the *body* of the rule. Literals are either *predicates* over *fields* (variables and constants), or function symbols applied to fields. The rules can refer to each other in a cyclic fashion to express recursion. The order in which the rules are presented in a program is semantically immaterial. The commas separating the predicates in a rule are logical conjuncts (*AND*); the order in which predicates appear in a rule body also has no semantic significance, though most implementations (including ours) employ a left-to-right execution strategy. The *query* specifies the output of interest.

The predicates in the body and head of traditional Datalog rules are relations, and we refer to them interchangeably as predicates, relations or tables. Each relation has a *primary key*, which consists of a set of fields that uniquely identify each tuple within the relation. In the absence of other information, the primary key is the full set of fields in the relation.

By convention, the names of predicates, function symbols and constants begin with a lower-case letter, while variable names begin with an upper-case letter. Most implementations of Datalog enhance it with a limited set of function calls (which start with "*f_*" in our syntax), including boolean predicates and arithmetic computations. Aggregate constructs are represented as functions with field variables within angle brackets (<>). For most of our discussion, we do not consider negated predicates; we return to the topic of negation as part of our future work (Chapter 10).

As an example, Figure 2.1 shows a Datalog program that computes the next hop along the shortest paths between all pairs of nodes in a graph. The program abbreviates some of its predicates as shown in Table 2.1. The program has four rules (which for convenience we label r1-r4), and takes as input a base ("extensional") relation *link(Source, Destination, Cost)*. Rules r1-r2 are used to derive "paths" in the graph, represented as tuples in the derived ("intensional") relation *path(S,D,Z,C)*. The *S* and *D* fields represent the source and destination endpoints of the path; *Z* contains the "next hop" in the graph along the shortest path that a node *S* should take in order to go to node *D*. The number and types of fields in relations are inferred from their (consistent) use in the program's rules.

Rule r1 produces *path* tuples directly from existing *link* tuples, and rule r2 recursively produces *path* tuples of increasing cost by matching (or *unifying*) the destination fields of existing links to the source fields of previously computed paths. The matching is expressed using the repeated "Z" variable in *link(S,Z,C1)* and *path(Z,D,Z2,C2)* of rule r2. Intuitively, rule r2 says that "if there is a link from node *S* to node *Z*, and there is a path from node *Z* to node *D*, then there is a path from node *S* to node *D* via *Z*".

Given the *path* relation, rule r3 derives the relation *spCost(S,D,C)* that computes the minimum cost *C* for each source (*S*) and destination (*D*) for all input paths. Rule r4 takes as input *spCost* and *path* tuples and then computes *shortestPathHop(S,D,Z,C)* tuples that contains the next hop (*Z*) along the shortest path from *S* to *D* with cost *C*. Last, the *Query* specifies the output of interest to be the *shortestPath* table.

2.2 Network Datalog by Example

Before diving into the formal definitions of *NDlog*, we first introduce *NDlog* by example using a distributed variant of the earlier *Shortest-Path-Hop* Datalog program. This dis-

```
r1 path(S,D,D,C) :- link(S,D,C).
r2 path(S,D,Z,C) :- link(S,Z,C1), path(Z,D,Z2,C2), C = C1 + C2.
r3 spCost(S,D,min<C>) :- path(S,D,Z,C).
r4 shortestPathHop(S,D,C) :- spCost(S,D,C), path(S,D,Z,C).
Query shortestPathHop(S,D,Z,C).
```

Figure 2.1: Shortest-Path-Hop Datalog program.

Predicate	**Schema**
link(S,D,C)	path(Source,Destination,Cost)
path(S,D,Z,C)	path(Source,Destination,NextHop,Cost)
spCost(S,D,C)	spCost(Source,Destination,Cost)
shortestPathHop(S,D,Z,C)	shortestPathHop(Source,Destination,NextHop,Cost)

Table 2.1: Predicates and the corresponding schemas used in the *Shortest-Path-Hop* Datalog program shown in Figure 2.1.

tributed *NDlog* program, shown in Figure 2.2, computes for every node, the next hop along the shortest paths of all nodes in a network in a distributed fashion. We use this *NDlog* program to highlight the following key points:

- *NDlog* builds upon traditional Datalog in order to meet four new requirements: *distributed computation*, *link-restricted communication*, *soft-state data and rules*, and *incremental maintenance of network state*.
- When this program is executed, the resulting communication and network state resembles the well-known *distance vector* and *path vector* routing protocols [92].
- This example program demonstrates the compactness of *NDlog*. In four *NDlog* rules, we are able to specify and implement a routing protocol widely used to compute shortest routes in a network.

```
materialize(#link,infinity,infinity,keys(1,2)).
materialize(path,infinity,infinity,keys(1,2,3,4)).
materialize(spCost,infinity,infinity,keys(1,2)).
materialize(shortestPathHop,infinity,infinity,keys(1,2)).
sh1 path(@S,D,D,C) :- #link(@S,D,C).
sh2 path(@S,D,Z,C) :- #link(@S,Z,C1), path(@Z,D,Z2,C2), C = C1 + C2.
sh3 spCost(@S,D,min<C>) :- path(@S,D,Z,C).
sh4 shortestPathHop(@S,D,Z,C) :- spCost(@S,D,C), path(@S,D,Z,C).
Query shortestPathHop(@S,D,Z,C).
```

Figure 2.2: Shortest-Path-Hop *NDlog* program.

2.2.1 Overview of NDlog

An *NDlog* program is largely composed of table declaration statements and rules; we consider each in turn. In *NDlog*, all input relations and rule derivations are stored in *materialized* tables. Unlike Datalog, tables must be defined explicitly in *NDlog* via *materialize* statements, which specify constraints on the size and lifetime of tuple storage – any relations not declared as tables are treated as named *streams* of tuples. Each *materialize(name, lifetime, size, primary keys)* statement specifies the relation name, lifetime of each tuple in the relation, maximum size of the relation, and fields making up the primary key of each relation[1]. If the primary key is the empty set *()*, then the primary key is the full set of fields in the relation. For example, in the *Shortest-Path-Hop NDlog* program, all the tables are specified with infinite sizes and lifetimes.

The execution of *NDlog* rules will result in the derivation of tuples that are stored in materialized tables. For the duration of program execution, these materialized results are

[1]We have a convention of starting the offset by 1 in the P2 system, as 0 is reserved in the implementation for the table name.

incrementally recomputed as the input relations are updated. For example, the update of *#link* tuples will result in new derivations and updates to existing *path*, *spCost* and *shortestPathHop* tuples. In addition, if an *NDlog* rule head is prepended with an optional keyword *delete*, the derived tuples are used to delete an exact match tuple in its relation instead.

Since network protocols are typically computations over distributed network state, one of the important requirements of *NDlog* is the ability to support rules that express distributed computations. *NDlog* builds upon traditional Datalog by providing control over the storage location of tuples explicitly in the syntax via *location specifiers*. Each location specifier is an attribute within a predicate that indicates the partitioning field of each relation. To illustrate, in Figure 2.2, each predicate in the *NDlog* rules has an "@" symbol prepended to a single field denoting the location specifier. Each tuple generated is stored at the address determined by its location specifier. For example, all *path* and *link* tuples are stored based on the address stored in the first field @*S*.

Interestingly, while *NDlog* is a language to describe networks, there are no explicit communication primitives. All communication is implicitly generated during rule execution as a result of data placement. For example, in rule sh2, the *path* and *#link* predicates have different location specifiers, and in order to execute the rule body of sh2 based on their matching fields, *link* and *path* tuples have to be shipped in the network. It is the movement of these tuples that will generate the messages for the resulting network protocol.

2.2.2 From Query Specifications to Protocol Execution

Having provide a high-level overview of *NDlog*, we demonstrate the execution of the *Shortest-Path-Hop NDlog* program via an example network shown in Figure 2.3. We show that the resulting communication and network state generated in program execution resembles the distance-vector protocol.

In our example network, each node is running the *Shortest-Path-Hop* program. For simplicity, we show only the derived paths along the solid lines even though the network connectivity is bidirectional (dashed lines). Our discussion is necessarily informal since we have not yet presented our distributed implementation strategies; in Chapter 5, we show in greater detail the steps required to generate the execution plan. Here, we focus on a high-level understanding of the data movement in the network during query processing.

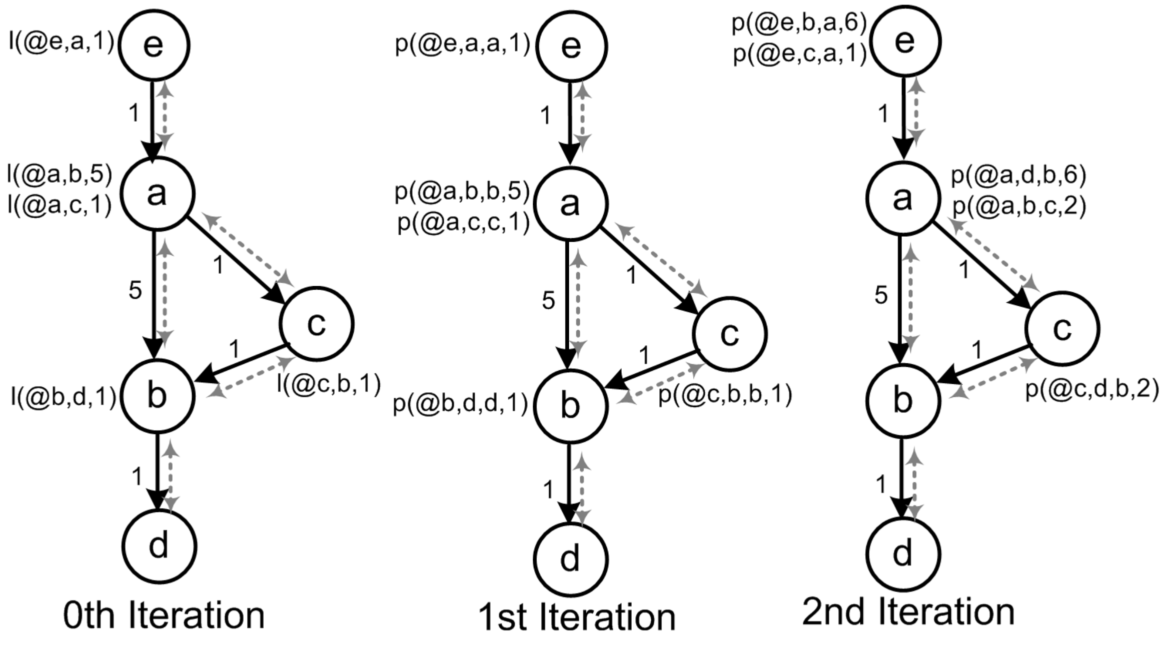

Figure 2.3: Shortest-Path program example execution. *l* and *p* are used as abbreviations for *link* and *path* respectively.

For our discussion here, we simplify communication by describing it in *iterations*, where at each iteration, each network node generates *paths* of increasing hop count, and then propagates these paths to neighbor nodes along links. Each *path* tuple contains the *nextHop* field, which indicates for each path the next hop to route the message in the network. In Figure 2.3, we show newly derived path tuples at each iteration. In the 1^{st} iteration, all nodes initialize their local *path* tables to 1-hop *path* tuples using rule sh1. In the 2^{nd}

iteration, using rule sh2, each node takes the input *path* tuples generated in the previous iteration, and computes 2-hop paths, which are then propagated to its neighbors. For example, *path(@a,d,b,6)* is generated at node *b* using *path(@b,d,d,1)* from the 1st iteration, and propagated to node *a*.

As *path* tuples are being computed and received at nodes, *spCost* and *shortestPathHop* tuples are also incrementally computed. For example, node *a* computes *path(@a,b,b,5)* using rule sh1, and then derives *spCost(@a,b,5)* and *shortestPathHop(@a,b,b,5)* using rules sh4-sh5. In the next iteration, node *a* receives *path(@a,b,c,2)* from node *c* which has lower cost compared to the previous shortest cost of 5, and hence the new tuples *spCost(@a,b,2)* and *shortestPathHop(@a,b,c,2)* replaces the previous values.

In the presence of path cycles, the *Shortest-Path-Hop* program never terminates, as rules sp1 and sp2 will generate paths of ever increasing costs. However, this can be fixed either by storing the entire path and adding a check for cycles within the rules. Alternatively, a well-known optimization (Section 8.1.1) can be used when costs are positive to avoid cycles. Intuitively, this optimization reduces communication overhead by sending only the *path* tuples that result in changes to the local *spCost* and *shortestPathHop* tables, hence limiting communication to only *path* tuples that contribute to the eventual shortest paths.

Algorithm 2.1 Pseudocode for the *Shortest-Path-Hop NDlog* program

```
$path(@Z,D,D,C) \leftarrow \#link(@Z,D,C)$                [Rule sh1]
while receive $< path(@Z,D,Z2,C2) >$
    foreach neighbor $\#link(@S,D,C)$                 [Rule sh2]
        send $path(@S,D,Z,C1+C2)$ to neighbor $@S$
    end
end
```

Figure 2.1 shows the pseudocode of executing rules sh1-sh2 from the perspective of a single node. Interestingly, the computation of the above program resembles the computation of the distance vector protocol [92] that is commonly used to compute shortest path routes in a network. In the distance vector protocol, each node advertises <*destination, path-cost*> information to all neighbors, which is similar to the *path* tuples exchanged by nodes at each iteration. All nodes use these advertisements to update their routing tables with the next hop along the shortest paths for each given destination. This is similar to computing new *shortestPathHop* tuples from *path* tuples using rules sh3-sh4. The main difference between our *NDlog* program and the actual distance vector computation is that rather than sending individual path tuples between neighbors, the traditional distance vector method batches together a vector of costs for all neighbors.

In the *Shortest-Path-Hop* program, the protocol only propagates the *nextHop* and not the entire path. In most practical network protocols such as the Border Gateway Protocol (BGP) [92], the entire path is included either for source routing or more commonly, to prevent infinite path cycles. This is typically known as the *path vector* protocol, where the *path vector* is the list of nodes from the source to the destination.

Figure 2.4 shows the *Shortest-Path NDlog* program that implements the path-vector protocol. The program is written with only minor modifications to the earlier *Shortest-Path-Hop NDlog* program. The program computes the entire path vector for a given source to destination, by adding an extra *pathVector* field in the *path* predicate that the full path. The function *f_init(X,Y)* initializes the path vector with nodes *X* and *Y*, and the function *f_concatPath(N,P)* prepend a node *N* to an existing path vector *P*. We revisit more examples of routing protocols in Chapter 3.

2.2.3 Other Requirements of NDlog

In addition to distributed computations, *NDlog* requires the following additional features for *link-restricted communication*, *soft-state data and rules*, and *incremental maintenance of network state*. We briefly describe them here, followed by more detailed descriptions in the rest of the chapter.

```
materialize(#link,infinity,infinity,keys(1,2)).
materialize(path,infinity,infinity,keys(4)).
materialize(spCost,infinity,infinity,keys(1,2)).
materialize(shortestPath,infinity,infinity,keys(1,2)).
sp1 path(@S,D,D,P,C) :- #link(@S,D,C), P = f_init(S,D).
sp2 path(@S,D,Z,P,C) :- #link(@S,Z,C1), path(@Z,D,Z2,P2,C2), C = C1 + C2,
                                    P = f_concatPath(S,P2).
sp3 spCost(@S,D,min<C>) :- path(@S,D,Z,P,C).
sp4 shortestPath(@S,D,P,C) :- spCost(@S,D,C), path(@S,D,Z,P,C).
Query shortestPath(@S,D,P,C).
```

Figure 2.4: Shortest-Path *NDlog* program.

- **Link-restricted communications:** In order to send a message in a low-level network, there needs to be a link between the sender and receiver. This is not a natural construct in Datalog. Hence, to model physical networking components where full-connectivity is not always available, *NDlog* provides syntactic restrictions that can be used to ensure that rule execution results in communication only among nodes that are physically connected. This is syntactically achieved with the use of the special *#link* predicate in all *NDlog* programs.

- **Soft-state data and rules:** In typical network protocols, the generated network state is maintained as *soft-state* [31] data. In the soft state storage model, stored data have a *lifetime* or time-to-live (TTL), and are deleted when the lifetime has expired. The soft state storage model requires periodic communication to refresh network state. Soft state is often favored in networking implementations because in a very simple manner it provides well-defined eventual consistency semantics. Intuitively, periodic refreshes to network state ensure that the eventual values are obtained even if there are transient errors such as reordered messages, node disconnection or link failures. While soft state is useful for maintaining distributed state, we also make extensive use of traditional *"hard-state"* data with infinite lifetimes for storing persistent counters, local machine state and archival logs.

- **Incremental maintenance of network state:** In practice, most network protocols are executed over a long period of time, and the protocol incrementally updates and repairs routing tables as the underlying network changes (link failures, node departures, etc). To better map into practical networking scenarios, one key distinction that differentiates the execution of *NDlog* from earlier work in Datalog is our support for continuous rule execution and results materialization, where all tuples derived from *NDlog* rules are materialized and incrementally updated as the underlying network changes. As in network protocols, such incremental maintenance is required both for timely updates and for avoiding the overhead of recomputing all routing tables "from scratch" whenever there are changes to the underlying network.

In the rest of this chapter, using the *Shortest-Path* program in Figure 2.4 as our primary example, we demonstrate the extensions to both the data and query model of traditional Datalog in order to handle the requirements of distributed computations (Section 2.3), link-restricted communications (Section 2.4), soft-state data and rules (Section 2.5), and incremental maintenance of network state (Section 2.6).

2.3 Distributed Computation

One novelty of our setting from a database perspective, is that data is distributed and relations may be partitioned across sites. *NDlog* gives the program writer *explicit* control of data placement with the use of *location specifiers* in each predicate:

Definition 2.1 A *location specifier* is a field in a predicate whose value per tuple indicates the network storage location of that tuple.

The location specifier field is of type address having a value that represents a network location. It is used as a partitioning field for its table across all nodes in the network, similar to horizontally partitioned tables in distributed databases [90]. We require that each predicate has a single location specifier field that is notated by an "@" symbol. Each predicate has exactly one field as its location specifier. For example, the location specifier of *#link(@S,D,C)* is *@S*. This means that all *#link* tuples are stored based on the address value of the *@S* field.

Given that predicates have location specifiers, we can now distinguish *local* and *distributed* rules:

Definition 2.2 *Local rules* are rules that have the same location specifier in each predicate, including the head.

We refer to non-local rules as *distributed rules*. Local rules can be executed without any distributed logic. In the *Shortest-Path* program, rules sp1, sp3 and sp4 are local, while sp2 is a distributed rule since the *#link* and *path* body predicates are stored at different locations.

2.4 Link-Restricted Communication

In real networking components such as routers, switches and autonomous routing systems on the Internet, each node is connected to relatively few other nodes for communication. To model actual physical links connecting nodes on which communication can happen, we introduce the concept of a *link relation*, which is defined as follows:

Definition 2.3 A *link relation* link(src,dst,...) represents the connectivity information of the network being queried.

The first two fields of each link table entry contain the source and destination addresses of a network link respectively, followed by an arbitrary number of other fields (typically metrics) describing the link.

Definition 2.4 A *link literal* is a link relation that appears in the body of a rule prepended with the "#" symbol.

In the *Shortest-Path* program, the link literal appears in the rule body of non-local rules sp1 and sp2. We can now define a simple syntactic constraint on the rules to ensure that communication for all distributed rules takes place only along the physical links:

Definition 2.5 A *link-restricted* rule is either a local rule, or a rule with the following properties:

- There is exactly one link literal in the body
- All other literals (including the head predicate) have their location specifier set to either the first (source) or second (destination) field of the link literal.

This syntactic constraint precisely captures the requirement that we are able to operate directly on a network whose link connectivity is not a full mesh. For example, rule sp2 of Figure 2.4 is link-restricted but has some relations whose location specifier is the source @S, and others whose location specifier is the destination @D. As we shall demonstrate in Chapter 5, all link-restricted rules are rewritten into a canonical form where every rule body can be evaluated on a single node. In addition, all communication for each rewritten rule only involves sending messages along links.

2.5 Soft-state Data and Rules

In this section, we define types of relations and rules based on *hard-state* and *soft state* storage models. Note that these definitions are orthogonal to our earlier definitions on distributed and link-restricted data and rules. For example, a soft-state relation (rule) can be link-restricted and/or distributed.

```
materialize(#link,10,infinity,keys(1,2)).
materialize(pingRTT,10,5,keys(1,2)).
materialize(pendingPing,10,5,keys(1,2)).
pp1 ping(@S,D,E) :- periodic(@S,E,5), #link(@S,D).
pp2 pingMsg(S,@D,E) :- ping(@S,D,E), #link(@S,D).
pp3 pendingPing(@S,D,E,T) :- ping(@S,D,E), T = f_now().
pp4 pongMsg(@S,E) :- pingMsg(S,@D,E), #link(@D,S).
pp5 pingRTT(@S,D,RTT) :- pongMsg(@S,E), pendingPing(@S,D,E,T),
                         RTT = f_now() - T.
pp6 #link(@S,D) :- pingRTT(@S,D,RTT).
Query pingRTT(@S,D,RTT).
```

Figure 2.5: Ping-Pong *NDlog* program.

Predicate	**Schema**
#link(@S,D,E)	#link(@Source,Destination,EventID)
ping(@S,D,E)	ping(@Source,Destination,EventID)
pingMsg(S,@D,E)	pingMsg(Source,@Destination,EventID)
pongMsg(@S,E)	pongMsg(@Source,EventID)
pendingPing(@S,D,E,T)	pendingPing(@Source,Destination,EventID,Time)
pingRTT(@S,D,RTT)	pingRTT(@Source,Destination,RoundTripTime)

Table 2.2: Schema of tables and events used in the *Ping-Pong* program

In the rest of this section, we use the *Ping-Pong* program in Figure 2.5 to illustrate *NDlog* rules that manipulate soft-state data. The *Ping-Pong* program implements a simple ping program where each node periodically pings its neighbor nodes to computes the round-trip time (RTT). Unlike the earlier *Shortest-Path* program, all relations used in the *Ping-Pong* program are declared with finite lifetimes and sizes. There are also some relations such as *ping*, *pingMsg* and *pongMsg* that are not declared using the *materialize* keyword. These relations are known as *event relations*, and they consist of zero-lifetime tuples that are used to execute rules but are not stored.

Rule pp1 is triggered periodically using the special *periodic* predicate. The *periodic(@S,E,5)* predicate denotes an infinite stream of *periodic* event tuples generated at node *S* every 5 seconds with random identifier *E*. This allows rule pp1 to generate at 5 seconds interval, a *ping(@S,D,E)* event tuple at source node *S* to all its neighbors with destination *D*. Each *ping* is uniquely identified with an event identifier *E*. Each *ping* tuple is then used to generate a *pingMsg(S,@D,E)* tuple that is sent to destination node *D* (rule pp2). A *pendingPing(@S,D,E,T)* tuple is also stored locally to record the creation time *T* of *ping(@S,D,E)*.

In rule pp4, whenever a node *D* receives a *pingMsg(S,@D,E)* tuple from the source node *S*, it replies with a *pongMsg(@S,E)* tuple to node *S*. Upon receiving the *pingMsg(@S,E)* tuple, rule pp5 is used by node *S* to compute the RTT between itself and node *D* based on the time recorded in *pendingPing(@S,D,E,T)*. A successful reply to a ping message indicates that the neighbor is alive. This results in the refresh of #link tuples in rule pp6.

2.5.1 Hard-state vs Soft-state Data

In *NDlog*, we distinguish between hard-state and soft-state relations based on the lifetime parameter in *materialized* statements.

Definition 2.6 A *hard-state relation* is one that is materialized with infinite lifetime.

Hard-state relations are similar to data stored in traditional databases, which are non-expiring and have to be deleted explicitly. The *#link* relation in the *Shortest-Path* program is an example of a hard-state relation. All *#link* tuples persist unless explicitly deleted. For derivations such as *path* in the *Shortest-Path-Hop* program, there can be multiple derivations for the same tuple. Hence, we need to keep track of all such derivations for hard-state relations until all derivations are invalidated due to deletions.

Definition 2.7 A *soft-state relation* is one that is materialized with finite lifetime.

Tuples that are inserted into soft-state tables are stored only for the duration of the table's lifetime. If required by the network protocol, these soft-state tuples can be refreshed via *NDlog* rules. Unlike hard-state relations, we do not need to keep track of multiple derivations of the same tuple. Instead, a *refresh* occurs when the same tuple is inserted into the table, resulting in the extension of the tuple by its specified lifetime. For example, the *#link* relation in the *Ping-Pong* program is a soft-state relation, and all *#link* tuples generated are deleted after ten seconds unless they are refreshed by rule pp6 before they expire.

Definition 2.8 An *event relation* is a soft-state relation with zero lifetime.

Event relations can either be declared explicitly via *materialize* statements with the lifetime parameter set to 0, or implicitly if they are not declared in any *materialize* statements. Event relations are typically used to represent message "streams" (*e.g.*, *pingMsg*, *pongMsg* in the *Ping-Pong* program), or periodically generated local events via a built-in *periodic* predicate (*e.g.*, in rule pp1):

Definition 2.9 The *periodic(@N,E,T,K)* event relation is a built-in relation that represents an stream of event tuples generated at node *N* every *T* seconds (up to an optional *K* times) with a random event identifier *E*. If *K* is omitted, the stream is generated infinitely.

Built-in streams in NDLog are akin to the *foreign functions* [1] of LDL++ [9] or the table functions of SQL, but their storage semantics are those of events, as described above. For example, the *periodic(@S,E,5)* in rule pp1 denotes an infinite stream of *periodic* event tuples generated at node *S* every 5 seconds with random identifier *E*.

2.5.2 Hard-state and Soft-state Rules

Following our definitions of hard-state and soft-state data, we present in this section *hard-state rules* and *soft-state rules*, which differs on their use of hard-state and soft-state relations in the rules:

Definition 2.10 A *hard-state rule* contains only hard-state predicates in the rule head and body.

Definition 2.11 A *soft-state rule* contains at least one soft-state predicate in the rule head or body.

We further classify soft-state rules as follows:

Definition 2.12 A *pure soft-state rule* has a soft-state predicate in the rule head, and at least one soft-state predicate in the rule body.

Definition 2.13 A *derived soft-state rule* has a soft-state predicate in the rule head, but only hard-state predicates in the rule body.

Definition 2.14 An *archival soft-state rule* has a hard-state rule head, and at least one soft-state predicate in the rule body.

Archival soft-state rules are primary used for archival or logging purposes. These rules derive hard-state tuples that persist even after the input soft-state input tuples that generate them have expired.

Since event relations are considered soft-state relations (with zero lifetimes), they can be used in any of the three soft-state rules above. During rule execution, input event tuples persist long enough for rule execution to complete and are then discarded. Since they are not stored, *NDlog* does not model the possibility of two instantaneous events occurring simultaneously. Syntactically, this possibility is prevented by allowing no more than one event predicate in soft-state rule bodies:

Definition 2.15 An *event soft-state rule* is a soft-state rule with *exactly one* event predicate in the rule body.

Using the *Ping-Pong* program as our example, all rules are pure soft-state relations since we do not involve any hard-state relations in this program. In addition, rules pp1-pp5 are event soft-state rules that take as input one event predicate (*periodic*, *ping*, *ping*, *pingMsg* and *pongMsg* respectively).

2.6 Incremental Maintenance of Network State

As in network protocols, *NDlog* rules are designed to be executed over a period of time and incrementally updated based on changes in the underlying network. During rule execution, depending on their specified lifetimes, all derived tuples are either stored in materialized table or generated as events. All materialized derivations have to be incrementally recomputed by long-running *NDlog* rules in order to maintain consistency with changes in the input base tables.

For hard-state rules, this involves the straightforward application of traditional materialized view maintenance techniques [54]. We consider three types of modifications to hard-state relations: insertions of new tuples, deletions of existing tuples, and updates (which can be modeled as deletion followed by an insertion). Note that inserting a tuple where there is another tuple with the same primary key is considered an update, where the existing tuple is deleted before the new one is inserted.

Similar to traditional database materialized views, the deletions of any input relations result in *cascaded deletions*, where a deleted tuple may *cascade* and lead to the deletion of previously derived tuples. For example, whenever a *#link* tuple is deleted, all *path* tuples that are generated using this *#link* tuple have to be deleted as well. Since there can be multiple derivations of each unique tuple, we need to keep track of all of them and only delete a tuple when all its derivations are deleted.

The incremental maintenance of soft-state rules is carried out in a slightly different fashion due to the presence of soft-state relations. Two types of modifications are considered: insertions of new tuples or *refreshes* of existing soft-state tuples. Recall from Section 2.5.1 that a *refresh* occurs when the same tuple is inserted into the table, resulting in the extension of the tuple by its specified lifetime. These soft-state refreshes in turn lead to *cascaded refreshes*, where previously derived soft-state tuples are rederived and hence also refreshed. Unlike the maintenance of hard-state rules, cascaded deletions do not occur in soft-state rules. Instead, all derived soft-state tuples are stored for their specified lifetimes and timeout in a manner consistent with traditional soft-state semantics.

2.7 Summary of Network Datalog

Given these preliminaries, we are now ready to present *NDlog*. The *NDlog* data model is based on the relational model with the following constraints:

1. All *NDlog* relations are horizontally partitioned in the network based on the location

specifier attribute.

2. One of the *NDlog* relations, denoted by *#link(src,dst,...)* represents the connectivity information of the network being queried.

3. A *NDlog* relation is either a hard-state or soft-state relation depending on its lifetime.

A *NDlog* program is a Datalog program that satisfies the following syntactic constraints:

1. All predicates in an *NDlog* rule head or rule body have a location specifier attribute.

2. Any distributed *NDlog* rules in the program are link-restricted by some link relation.

3. A *NDlog* rule is either a hard-state or soft-state rule.

In addition, the results of executing *NDlog* rules are materialized for their table lifetimes, and incrementally maintained as described in Section 2.6.

2.7.1 Discussion

Interestingly, *NDlog* uses a somewhat more physical data model than the relational model, and a correspondingly somewhat more physical language. The main reason for doing this is to capture the essence of a network protocol – communication over links – in a way that remains largely declarative, leaving significant latitude for a compiler to choose an implementation of the specification. Note that most aspects of a program other than storage location and communication pairs are left unspecified – this includes the order in which tuples of a set are handled and the order in which predicates of a rule are considered. In addition, the need for partitioning via location specifiers and link restriction reflects low-level networks. In principle, given a network implemented in this manner to achieve all-pairs communication, higher-level logic could be written without reference to locations or links. This is a natural extension to *NDlog*, but since this dissertation focuses on networking, we do not explore it further.

2.8 Summary

In this chapter, we formally defined the *NDlog* language. Our language is based on Datalog, and we described the extensions to address *NDlog*'s four main requirements of *distributed computation*, *link-restricted communication*, *support for soft-state data and rules*, and *incremental maintenance of network state*. All of these extensions have been motivated by the distributed settings we target in declarative networking, which are a departure from the environments in which traditional Datalog was used. In subsequent sections, we provide two concrete instances of declarative networking, namely *declarative routing* and *declarative overlays*, and also describe in detail how *NDlog* programs are processed and executed to implement the network protocols.

Chapter 3

Declarative Routing

Having given an overview of the *NDlog* language, this chapter focuses on *declarative routing*: the declarative specification of routing protocols for building extensible routing infrastructures. Declarative networking aims to strike a better balance between the extensibility and the robustness of a routing infrastructure. In addition to being a concise and flexible language for routing protocols, we show that *NDlog* is amenable to static analysis, making it an attractive language for building safe, extensible routing infrastructures.

The chapter is organized as follows. First, we present the motivation of declarative routing in Section 3.1. Next, we provide an overview of our execution model in Section 3.2. We illustrate the flexibility of *NDlog* through several declarative routing examples in Section 3.3. We then address the challenges of security in Section 3.4, and route maintenance under dynamic networks in Section 3.5.

3.1 Motivation

Designing routing protocols is a difficult process. This is not only because of the distributed nature and scale of the networks, but also because of the need to balance the extensibility and flexibility of these protocols on one hand, and their robustness and efficiency on the other hand. One need look no further than the Internet for an illustration of these different tradeoffs.

Today's Internet routing protocols, while arguably robust and efficient, are hard to change to accommodate the needs of new applications such as improved resilience and higher throughput. Upgrading even a single router is hard [57]. Getting a distributed routing protocol implemented correctly is even harder. And in order to change or upgrade a deployed routing protocol today, one must get access to *each* router to modify its software. This process is made even more tedious and error prone by the use of conventional programming languages that were not designed with networking in mind.

Several solutions have been proposed to address the lack of flexibility and extensibility in Internet routing. Overlay networks allow third parties to replace Internet routing with new, "from-scratch" implementations of routing functionality that run at the application layer. However, overlay networks simply move the problem from the network to the application layer where third parties have control: implementing or updating an overlay routing protocol still requires a complete protocol design and implementation, and requires access to the overlay nodes.

On the other hand, a radically different approach, *active networks* [119], allows network packets to modify the operation of networks by allowing routers to execute code within active network packets. This allows new functionality to be introduced to existing active networks without the need to have direct access to routers. However, due to the general programming models proposed for active networks, they present difficulties in both performance and the security and reliability of the resulting infrastructure.

In this chapter, we demonstrate that declarative routing provides a new point in this design space that aims to strike a better balance between the extensibility and the robustness of a routing infrastructure. With declarative routing, a routing protocol is implemented by writing a simple *NDlog* program, which is then executed in a distributed fashion at some or all of the nodes. Declarative routing can be viewed as a restrictive instantiation of active networks which aims to balance the concerns of expressiveness, performance and security, properties which are needed for an extensible routing infrastructure to succeed.

Declarative routing could evolve to be used in a variety of ways. One extreme view of the future of routing is that individual end-users (or their applications) will explicitly request routes with particular properties, by submitting route construction *NDlog* programs to the network. The safety and simplicity of declarative specifications would clearly be beneficial in that context. A more incremental view is that an administrator at an ISP might reconfigure the ISP's routers by issuing an *NDlog* program to the network; different *NDlog* programs would allow the administrator to easily implement various routing policies between different nodes or different traffic classes. Even in this managed scenario, the simplicity and safety of declarative routing has benefits over the current relatively fragile approaches to upgrading routers. While this second scenario is arguably the more realistic one, in this chapter, we consider the other extreme in which any node (including end-hosts) can issue an *NDlog* program. We take this extreme position in order to explore the limits of our design.

3.2 Execution Model

We model the routing infrastructure as a directed graph, where each link is associated with a set of parameters (*e.g.*, loss rate, available bandwidth, delay). The router nodes in the routing infrastructure can either be IP routers or overlay nodes.

Figure 3.1 shows the architecture of a typical declarative router. Like a traditional router, a declarative router maintains a *neighbor table*, which contains the set of neighbor routers that this router can forward messages to, and a *forwarding table* in the *forward-*

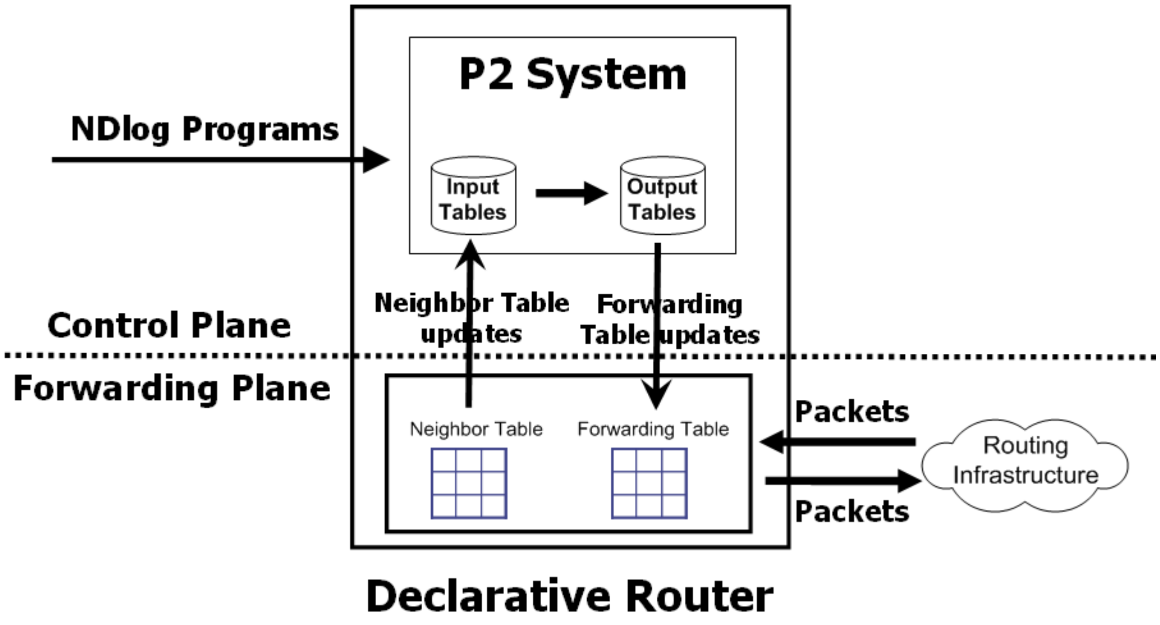

Figure 3.1: A Declarative Router.

ing plane, that is used to route incoming packets based on their destination addresses to neighboring nodes along a computed path path.

The forwarding table is created by the routing protocol that executes on the *control plane* of each router. Each routing protocol takes as input any updates to the local neighbor table, and implements a distributed computation where routers exchange route information with neighboring routers to compute new routes.

In a declarative router, a P2 runtime system runs on the control plane and takes as input local routing information such as the neighbor table. Instead of running a single routing protocol, the P2 system allows any routing protocols expressed in *NDlog* to be executed in a distributed fashion in the network. The results of the program are used to establish router forwarding state which the routers use for forwarding data packets. Alternatively, the computed results can be sent back to the party that issued the *NDlog*, which can use these results to perform source routing. Note that while the P2 system is used on the control plane in declarative routing, it can be used more generally on the forwarding plane as well, as we demonstrate in Chapter 6.

NDlog program dissemination and execution can happen in a variety of ways. In static scenarios, the program may be "baked in" to another artifact – *e.g.*, router firmware or peer-to-peer application software that is bundled with the P2 system. More flexibly, the program could be disseminated upon initial declaration to all or some of the nodes running the P2 system. It may be sufficient to perform dissemination via flooding, particularly if the program will be long-lived, amortizing the cost of the initial flood. As an optimization, instead of flooding the program in the network, we can instead "piggy-back" dissemination onto program execution: the program can be embedded into the first data tuple sent to each neighboring node as part of executing the *NDlog* program.

This execution model is based on a *fully distributed* implementation, where routes are computed in a decentralized fashion. As an alternative, in a *centralized* design such as the Routing Control Platform [44], network information is periodically gathered from the routing infrastructure, and stored at one or more central servers. Each program is sent to one or more of these servers, which process the programs using their internal databases and set up the forwarding state at the routers in the network.

During the execution of *NDlog* program, the neighbor table is periodically updated in response to link failures, new links, or link metric changes. These updates are performed by the routers themselves using standard mechanisms such as periodic pings. The P2 system is then notified of updates to the neighbor table, and will incrementally recompute entries into the forwarding table. In our discussion, this simple interface is the only interaction required between the P2 system and the router's core forwarding logic.

3.3 Routing Protocols By Examples

To highlight the flexibility of *NDlog*, we provide several examples of useful routing protocols expressed as *NDlog* rules. Our examples range from well-known routing protocols (distance vector, dynamic source routing, multicast, etc.) to higher-level routing concepts such as QoS constraints. This is by no means intended to be an exhaustive coverage of the possibilities of our proposal. Our main goal here is to illustrate the natural connection between recursive programs and network routing, and to highlight the flexibility, ease of programming, and ease of reuse afforded by a declarative language. We demonstrate that routing protocols can be expressed in a few *NDlog* rules, and additional protocols can be created by simple modifications).

3.3.1 Best-Path Routing

We start from the base rules sp1 and sp2 used in our first *Shortest-Path* program from the previous chapter. That example computes *all-pairs shortest paths*. In practice, a more common program would compute *all-pairs best paths*. By modifying rules sp2, sp3 and sp4, the *Best-Path* program in Figure 3.2 generalizes the all-pairs shortest paths computation, and computes the best paths for any path metric *C*:

```
bp1 path(@S,D,D,P,C) :- #link(@S,D,C), P=f_init(S,D).
bp2 path(@S,D,Z,P,C) :- #link(@S,Z,C1), path(@Z,D,Z2,P2,C2),
                                   C = f_compute(C1,C2), P = f_concatPath(S,P2).
bp3 bestPathCost(@S,D,AGG<C>) :- path(@S,D,Z,P,C).
bp4 bestPath(@S,D,P,C) :- bestPathCost(@S,D,C), path(@S,D,Z,P,C).
Query bestPath(@S,D,P,C).
```

Figure 3.2: Best-Path Program.

We have left the aggregation function *(AGG)* unspecified. By changing *AGG* and the function *f_compute* used for computing the path cost *C*, the *Best-Path* program can generate best paths based on Any metric including link latency, available bandwidth and node load. For example, if the program is used for computing the shortest paths, *f_sum* is the appropriate replacement for *f_compute* in rule bpr1, and *min* is the replacement for *AGG*. The resulting bestPath tuples are stored at the source nodes, and are used by end-hosts to perform source routing. Instead of computing the best path between any two nodes, this program can be easily modified to compute *all* paths, *any* path or the *Best-k* paths between any two nodes.

To avoid generating path cycles, we can add an extra predicate *f_inPath(P2,S)=false* to rule bp2 to avoiding computing best paths with cycles (*e.g.*, when computing the longest latency paths). We can further extend the rules from the *Best-Path* program by including constraints that enforce a QoS requirement specified by end-hosts. For example, we can restrict the set of paths to those with costs below a loss or latency threshold *k* by adding an extra constraint *C¡k* to the rules computing *path*.

3.3.2 Distance-Vector Routing

```
dv1 hop(@S,D,D,C) :- #link(@S,D,C).
dv2 hop(@S,D,Z,C) :- #link(@S,Z,C1), hop(@Z,D,W,C2), C = f_compute(C1,C2).
dv3 bestHopCost(@S,D,AGG<C>) :- hop(@S,D,Z,C).
dv4 bestPathHop(@S,D,Z,C) :- hop(@S,D,Z,C),bestHopCost(@S,D,C).
Query bestPathHop(@S,D,Z,C).
```

Figure 3.3: Distance-Vector Program.

Figure 3.3 shows a program that expresses the distance vector protocol for customized best routes for any given path metric. Rules dv1 and dv2 are modified from rules *bp1* and *bp2* from our previous example to generate the *hop* tuple that maintains only the next hop on the path, and not the entire path vector *P* itself[1]. Rules dv3 and dv4 are added to set up routing state in the network: *bestPathHop(@S,D,Z,C)* is stored at node *S*, where *Z* is the next hop on the best path to node *D*.

The distance vector protocol has the count-to-infinity problem [92], where link failures may result in long (sometimes infinite) protocol convergence times. By making a modification to rule dv2 and adding rule dv5, we can apply the well-known *split-horizon with poison reverse* [92] fix to this problem:

```
#include(dv1,dv3,dv4)
dv2 hop(@S,D,Z,C) :- #link(@S,Z,C1), hop(@Z,D,W,C2), C = C1 + C2, W != S.
dv5 hop(@S,D,Z,infinity):- #link(@S,Z,C1), hop(@Z,D,S,C2).
Query bestPathHop(@S,D,Z,C).
```

Figure 3.4: Distance-Vector Program with count-to-infinity fix in *NDlog*.

#include is a macro used to include earlier rules. Rule dv2 expresses that if node *Z* learns about the path to *D* from node *S*, then node *Z* does not report this path back to to *S*. Rule dv5 expresses that if node *Z* receives a path tuple with destination *D* from node *S*, then node *Z* will send a path with destination *D* and infinite cost to node *S*. This ensures that node *S* will not eventually use *Z* to get to *D*.

3.3.3 Policy-Based Routing

Our previous examples all illustrate a typical network-wide routing policy. In some cases we may want to restrict the scope of routing, *e.g.*, by precluding paths that involve "unde-

[1]The *W* field in dv2 represents the next-hop to node *D* from intermediate node *Z*, and can be ignored by node *S* in computing its next hop to node *D*.

sirable" nodes. An example would be finding a path among nodes in an overlay network on PlanetLab that avoids nodes belonging to untruthful or flaky ISPs. Such policy constraints can be simply expressed by adding an additional rule:

```
#include(bp1,bp2) pbr1 permitPath(@S,D,Z,P,C) :- path(@S,D,Z,P,C),
                                    excludeNode(@S,W), f_inPath(P,W)=false.
Query permitPath(@S,D,P,C).
```

Figure 3.5: Policy-Based Routing Program.

In this program, we introduce an additional table *excludeNode*, where *excludeNode(@S,W)* is a tuple that represents the fact that node *S* does not carry any traffic for node *W*. This table is stored at each node *S*.

If rules bp1 and bp2 are included as rules, we can generate bestPath tuples that meet the above policy. Other policy based decisions include ignoring the paths reported by selected nodes or insisting that some paths have to pass through (or avoid) one or multiple predetermined set of nodes.

3.3.4 Dynamic Source Routing

All of our previous examples use what is called *right* recursion, since the recursive predicates (*e.g.*, *path* in the rules sp2, bp2 and dv2) appears to the right of the matching *link*. Given that predicates are executed in a left-to-right order, the program semantics do not change if we flip the order of *path* and *link* in the body of these rules, but the execution strategy does change. In fact, using *left recursion* as follows, we implement the Dynamic Source Routing (dsr) protocol [64]:

Rule bp1 produces new one-hop paths from existing link tuples as before. Rule dsr2 matches the destination fields of newly computed path tuples with the source fields of link

```
#include(bp1,bp3,bp4)
dsr2 path(@S,D,Z,P,C) :- path(@S,Z,W,P1,C1), #link(@Z,D,C2),
                         C = f_compute(C1,C2), P = f_concatPath(P1,D).
Query bestPath(@S,D,P,C).
```

Figure 3.6: Dynamic Source Routing Program.

tuples. This requires newly computed path tuples be shipped by their destination fields to find matching links, hence ensuring that each source node will recursively follow the links along all reachable paths. Here, the function *f_concatPath(P,D)* returns a new path vector with node *D* appended to *P*. These rules can also be used in combination with bpr1 and bpr2 to generate the best paths. By adding two extra rules not shown here, we can also express the logic for sending each path on the reverse path from the destination to the source node.

3.3.5 Link State

To further illustrate the flexibility of our approach, we consider a link-state protocol that moves route information around the network very differently from the best-path variants. The following *Link-State* program expresses the flooding of links to all nodes in the network:

```
ls1 floodLink(@S,S,D,C,S) :- #link(@S,D,C).
ls2 floodLink(@M,S,D,C,N) :- #link(@N,M,C1), floodLink(@N,S,D,C,W), M != W.
Query floodLink(@M,S,D,C,N)
```

Figure 3.7: Link-State Program.

floodLink(@M,S,D,C,N) is a tuple storing information about *#link(@S,D,C)*. This tuple is flooded in the network starting from source node *S*. During the flooding process, node

M is the current node it is flooded to, while node *N* is the node that forwarded this tuple to node *M*.

Rule ls1 generates a *floodLink* tuple for every link at each node. Rule ls2 states that each node *N* that receives a *floodLink* tuple recursively forwards the tuple to all neighbors *M* except the node *W* that it received the tuple from. *NDlog* is based on the relational model that utilizes set computations, where duplicate tuples are not considered for computation twice. This ensures that no similar *floodLink* tuple is forwarded twice.

Once all the links are available at each node, a local version of the *Best-Path* program in Figure 3.2 is then executed locally using the *floodLink* tuples to generate all the best paths.

3.3.6 Multicast

The examples we have given so far support protocols for unicast routing. Here, we demonstrate a more complex example, using *NDlog* to construct a multicast dissemination tree from a designated root node to multiple destination nodes that "subscribe" to the multicast group. The following *Source-Specific-Multicast* program sets up such a forwarding tree rooted at a source node *a* for group *gid*:

```
#include(bp1,bp2,bp3,bp4)
m1 joinMessage(@I,N,P,S,G) :- joinGroup(@N,S,G), bestPath(@N,S,P1,C),
                              I = f_head(P1), P = f_tail(P1).
m2 joinMessage(@I,J,P,@S,G) :- joinMessage(@J,K,P1,S,G), I = f_head(P1),
                              P = f_tail(P1), f_isEmpty(P1) = false.
m3 forwardState(@I,@J,S,G) :- joinMessage(@I,J,P,S,G).
Query joinGroup(@N,a,gid)
```

Figure 3.8: Source-Specific-Multicast Program.

For simplicity of exposition, this program utilizes the *Best-Path* program (rules bp1, bp2, bp3, bp4) to compute the all-pairs best paths. We will discuss program optimization techniques to reduce the communication overhead for small multicast groups in Section 8.1.2.

Each destination node *n* joins the group *gid* with source *a* by issuing the program *joinGroup(@n,a,gid)*. This results in the generation of the following derived tuples:

- **joinMessage(@nodeID, prevNodeID, pathVector, source, gid)**. This tuple stores the multicast *join* message for group *gid*. It is sent by every destination node along its best path to the *@source* address of the group. At each intermediate node with address *nodeID*, we keep track of *prevNodeID*, which is the address of the node that forwarded this tuple. *pathVector* is the remaining path that this message needs to traverse in order to reach the source node.

- **forwardState(@nodeID, forwardNodeID, source, gid)**. This tuple represents source-specific state of the multicast dissemination tree at each intermediate node with address *nodeID*. If a message from *source* of multicast group *gid* is received at *nodeID*, it is forwarded to *forwardNodeID*.

Rules m1 and m2 create the *joinMessage* tuple at each participating destination node *N*, and forward this tuple along the best path to the source node *S*. Upon receiving a *joinMessage* tuple, rule M3 allows each intermediate node *I* to set up the forwarding state using the *forwardState(@I,J,S,G)* tuple. The predicate function *f_head(P)* returns the next node in the path vector *P*, and *f_tail(P)* returns the path vector *P* with the first node removed. *f_isEmpty(P)* returns true if *P* is empty.

Instead of a *source-specific* tree, with minor modifications, we can construct *core-based trees* [13]. Here, each participating node sends a *join* message to a designated *core* node to build a *shared* tree rooted at the core. Messages are then unicast to the core, which disseminates it using the shared tree.

3.4 Security Issues

Security is a key concern with any extensible system [115; 22]. In the network domain, this concern is best illustrated by active networks which, at the extreme, allow routers to download and execute arbitrary code.

Our approach essentially proposes *NDlog* as a Domain Specific Language (DSL) [125] for programming the control plane of a network. DSLs typically provide security benefits by having restricted expressibility. *NDlog* is attractive in this respect, both because of its strong theoretical foundations, and its practical aspects. *NDlog* rules written in the core[2] The core Datalog language have polynomial time and space complexities in the size of the input [6]. This property provides a natural bound on the resource consumption of *NDlog* programs.

However, many implementations of *NDlog* (including our own) augment the core language with various functions. Example of such functions are boolean predicates, arithmetic functions, and string or list manipulation logic (*e.g.*, *f_init*, *f_concatPath*, *f_inPath*, *f_isEmpty*, *f_head* and *f_tail*). With the addition of arbitrary functions, the time complexity of a *NDlog* program is no longer polynomial.

Fortunately, several powerful static tests have been developed to check for the termination of an augmented Datalog program on a given input [67]. In a nutshell, these tests identify recursive definitions in the program rules, and check whether these definitions terminate. Examples of recursive definitions that terminate are ones that evaluate monotonically increasing/decreasing predicates whose values are upper/lower bounded.

[2]Such a "core" language does not contain predicates constructed using function symbols.

The *NDlog* rules that pass these checks are general enough to express a large class of routing protocols. Thus, our augmented *NDlog* language offers a good balance between expressiveness and safety. We note that all the examples presented in this chapter pass such termination tests.

In addition, the execution of the program is "sandboxed" within the program engine. These properties prevent the program from accessing arbitrary router state such as in-flight messages, and the router's operating system state. As a result, *NDlog* eliminates many of the risks usually associated with extensible systems.

Of course, there are many other security issues beyond the safety of the *NDlog* language. Two examples are denial-of-service attacks and compromised routers. These problems are orthogonal to network extensibility, and we do not address them in this dissertation. We revisit them as part of our future work in Section 10.3.

3.5 Route Maintenance

During program execution, changes in the network might result in some of the computed routes becoming stale. These can be caused by link failures, or changes in the link metrics when these metrics are used in route computation. Ideally, the program should rapidly recompute a new route, especially in the case of link failures.

One solution is to simply recompute the programs from scratch, either periodically or driven by the party that has issued the programs. However, recomputing the program from scratch is expensive, and if done only periodically, the time to react to failures is a half-period on average.

The approach we employ in this dissertation is to utilize long-running or *continuous* queries that incrementally recompute new results based on changes in the network. To ensure incremental recomputations, all intermediate state of each program is retained in the program processor until the program is no longer required. This intermediate state includes any shipped tuples used in join computation, and any intermediate derived tuples.

As we discussed in Section 3.2, each declarative router is responsible for detecting changes to its local information or base tables and reporting these changes to its local program processor. These base tuple updates result in the addition of tuples into base tables, or the replacement of existing base tuples that have the same unique key as the update tuples. The continuous queries then utilize these updates and the intermediate state of rule executions to incrementally recompute some of their derived tuples.

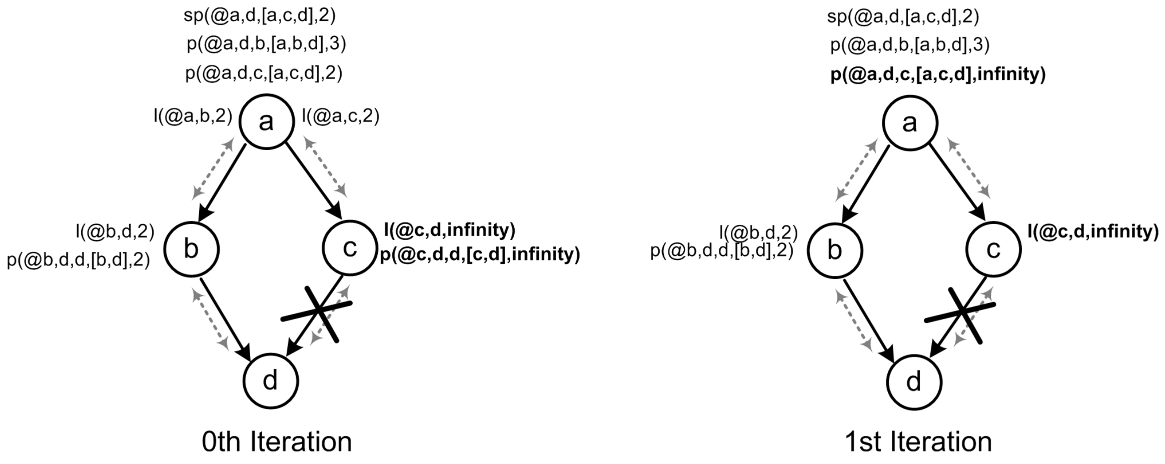

Figure 3.9: Derivation of alternative shortest path from node *a* to *d* when *#link(@a,b,1)* is deleted.

To illustrate, consider the *Shortest-Path* program that we introduce in Chapter 2. Figure 3.9 shows a simple four node network where all four nodes are running the *Shortest-Path* program. *l(@S,D,C)*, *p(@S,D,Z,P,C)* and *sp(@S,D,P,C)* abbreviates *link(@S,D,C)*, *path(@S,D,Z,P,C)* and *shortestPath(@S,D,P,C)* respectively.

Prior to the link failure, we assume that all shortest paths between all pairs have been computed. The figure shows the changes to the intermediate program states that led to the derivation of a new shortest path from node *a* to *d* when node *d* fails. For simplicity, we show only the derived paths along the solid lines even though the network connectivity is bidirectional (dashed lines). We denote an invalid path as one with infinite cost, although in practice, they are deleted from the *path* table. When *l(@c,d,1)* is deleted, the following steps are taken to derive *sp(@a,d,[a,b,d],3)*:

1. When neighbor *c* detects the failure of its link to *d* via a timeout, it generates an updated base tuple *l(@c,d,∞)* locally. This replaces the previous tuple *l(@c,d,1)*.

2. All one-hop paths at node *c* that traverse through *d* are set to infinite costs. For example, node *c* generates *p(@c,d,d,[c,d],∞)*.

3. *p(@c,d,d,[c,d],∞)* is joined with *l(@a,c,1)* to produce *p(@a,d,c,[a,c,d],∞)* which is sent to node *a*.

4. Upon receiving *p(@a,d,c[a,c,d],∞)*, node *a* computes a new shortest path *sp(@a,d,[a,b,d],3)*.

In this example, since we are computing the entire path vector, we can check for potential cycles. The failure is propagated hop-by-hop. Hence, the time taken for any update to converge is proportional to the network diameter, and bounded by the time it takes for a program to be executed from scratch.

Updates to link costs are handled in a similar fashion, except that rather than setting the costs to infinity, they are recomputed based on the new link costs. The updated paths may trigger further computation. For example, when the cost of paths are changed, rules bpr1 and bpr2 of the *Best-Path* program will generate alternative best paths accordingly.

In Chapter 5, we revisit in detail the processing of continuous queries using both hard-state and soft-state incremental view maintenance techniques [54].

3.6 Summary

In this chapter, we motivated declarative routing, as a means to permit flexible routing over the Internet. Through several examples, we demonstrate that the *NDlog* language is natural for expressing a wide variety of network routing protocols. Interestingly, we show that two important routing protocols (dynamic source routing and path vector protocols) differ only in the order in which predicates are evaluated. In Chapter 7 we measure the performance of declarative routing protocols such as the *Best-Path* program and validate that the scalability trends are similar to that of traditional approaches.

In the next two chapters, we describe how *NDlog* programs can be compiled into execution plans and executed using the P2 system to implement the routing protocols.

Chapter 4

P2 System Overview

Having presented a variety of declarative routing protocols using *NDlog*, in the next two chapters, we describe how *NDlog* programs can be compiled and executed to implement the network protocols. This chapter primarily focus on providing an overview of the P2 system, while the next chapter will focus specifically on the system component that processes *NDlog* programs. In Section 4.1, we present the architectural overview of the P2 declarative networking system and its different components. We then describe in Section 4.3 the runtime dataflow engine of P2 and compare P2 with alternative dataflow-based systems. In Section 4.4, we describe how network state is stored and managed as tables in the P2 system.

4.1 Architecture of P2

Figure 4.1 shows the architecture of the P2 declarative networking system from the perspective of a single node. There are three main components: the *planner*, the *dataflow installer* and the *dataflow engine*. The P2 system utilizes a dataflow framework at runtime for maintaining network state. P2 dataflows are similar to database query plans, which con-

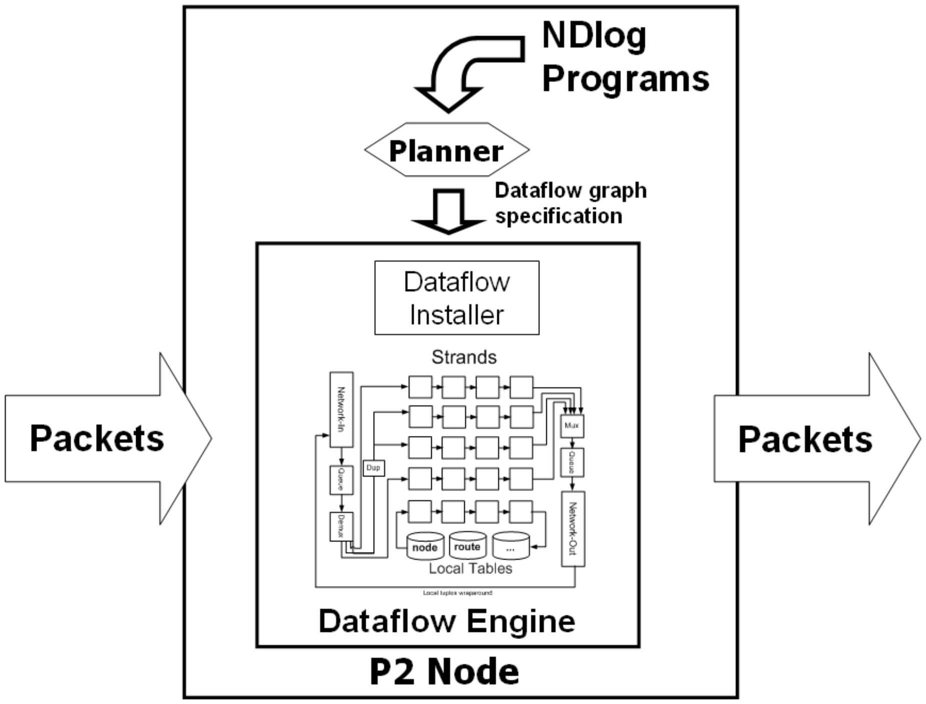

Figure 4.1: Components of a single P2 node.

sists of graphs that connect various database "operators" with dataflow edges that represent the passing of tuples among operators, possibly across a network.

To implement a network protocol, the planner takes as input the network specification expressed as a *NDlog* program, which is compiled into a dataflow graph. As an alternative to *NDlog*, the P2 system also provides a Python-based [95] scripting language that allows programmers to "hand-wire" dataflow graphs directly as input to the dataflow installer. In order to disseminate *NDlog* programs throughout a network, the P2 runtime system provides simple mechanisms for each node to send input *NDlog* programs by flooding to its neighbors. When a dataflow is installed, all the required *local tables* and indices necessary for the program are also created. Indices are created for every table's primary key, and additional indices are constructed on any table columns that are involved in unification (relational join). Once installed, dataflows are executed by the runtime engine until they are canceled.

The execution of the dataflow graph results in the implementation of the network protocol itself. The dataflow graph is registered locally at each node's dataflow engine via a *dataflow installer*. Each local dataflow participates in a global, *distributed* dataflow, with messages flowing among dataflows executed at different nodes, resulting in updates to the network state used by the network protocol. The distributed dataflow when executed performs the operations of a network protocol. The local tables store the state of the network protocols, and the flow of messages entering and leaving the dataflow constitute the network messages generated by the executing protocol.

4.2 P2 Dataflow Engine

The dataflow engine of P2 was inspired by prior work in both databases and networking. Software dataflow architectures like P2 occupy a constrained but surprisingly rich design space that has been explored in a variety of contexts[1]. Dataflow graphs have been used previously by parallel and distributed database query systems like Gamma [38], Volcano [51] and PIER [59] as their basic query executables.

The use of the dataflow framework has recently been explored in related work on extensible networks. For example, software router toolkits like Scout [87], Click [65] and XORP [56] in recent years have demonstrated that network message handling and protocol implementation can be neatly factored into dataflow diagrams. We adopt the Click term *element* for a node in a P2 dataflow graph, but as in database query plans, each edge in the

[1]There is also a rich hardware dataflow tradition in Computer Architecture (e.g., [91; 126]), with its own terminology and points of reference. For brevity, we do not consider those systems here, and when we refer to dataflow architectures, we limit our discussion to software dataflow.

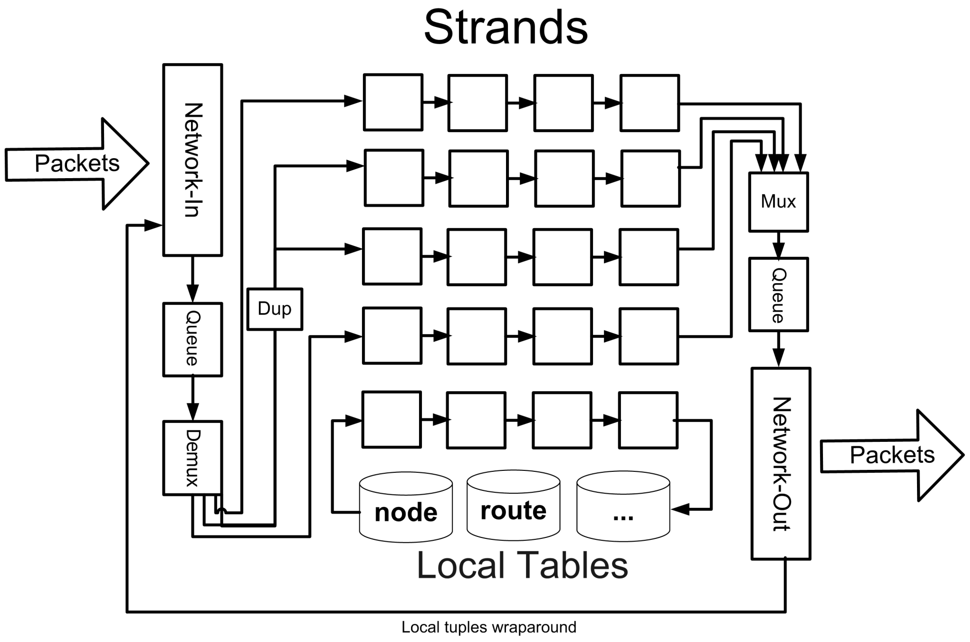

Figure 4.2: P2 Dataflow example at a single node.

graph carries a stream of well structured tuples, rather than annotated IP packets. Note that while all tuples flowing on a single edge share a structure (schema), tuples on one edge may have very different structure than tuples on another – this is a significant distinction with the uniform IP packets of Click.

Figure 4.2 shows an example of a P2 dataflow being executed at a single node. At the edges of the dataflow, we have a chain of network packet processing elements (encapsulated in the figure as *Network-In* and *Network-Out*) that are used to process incoming and outgoing messages respectively. Figure 4.3 shows an example implementation of the networking-related elements. Both the *Network-In* and *Network-Out* portion of the dataflow comprise a longer sequence of network-related elements that implement functionality for sending and receiving messages (*UDP-Tx* and *UDP-Rx*), and may also perform reliable transmission (*Retry* and *Ack*), and congestion control (*CC-Tx* and *CC-Rx* elements). These elements can be dynamically adapted (reordered, added or removed from the dataflow) based on the

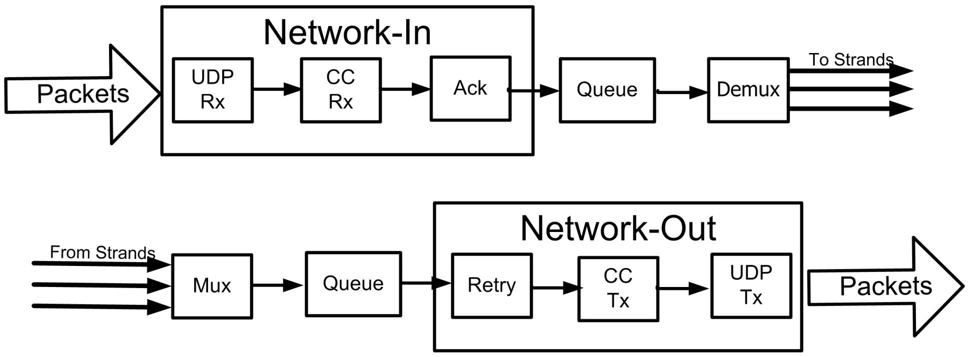

Figure 4.3: Example of expanded *Network-In* and *Network-Out* elements.

requirements of the declarative network (see [123]).

Messages that arrive into the dataflow are buffered using queues, and demultiplexed (using the *Demux* element) via the relation name of each tuple into *strands*, and then duplicated (using the *Dup* element) into multiple strands that require input from the same relation. The strands are directly compiled from our *NDlog* rules and implement the "logic" of the network. Each strand consists of a chain of elements implementing relational database operators like joins, selections, projections and aggregations. The use of joins is endemic to P2 because of our choice of *NDlog*: the unification (matching) of variables in the body of a rule is implemented in a dataflow by an equality-based relational join (*equijoin*). As shown in Figure 4.2, these strands take as input tuples that arrive via the network (output from the *Dup* element), local table updates (directly from the local tables) or local periodically generated events. The execution of strands either results in local table updates, or the sending of message tuples.

On the other side of the graph (shown as the *Network-Out* elements), message tuples are merged by a *Mux* element, queued and then sent based on their network destinations. Remote tuples are sent via an output queue to the network stack to be packetized, marshaled, and buffered by P2's UDP transport, while tuples destined for local consumption are "wrapped around" to the *Network-In* element and queued along with other input tuples arriving over the network.

In the P2 runtime, the *Network-In* and *Network-Out* elements can be shared by multiple overlays that are running concurrently. The P2 system will compile them into a single dataflow for execution, where the *Network-In* and *Network-Out* elements will be shared among the different overlays.

4.3 Dataflow Framework Implementation

We based our design in large part on our side-by-side comparison between the PIER peer-to-peer query engine [59] and the Click router [65]. Like PIER, P2 can manage structured data tuples flowing through a broad range of query processing operators, which may accumulate significant state and perform substantial asynchronous processing. Like Click, P2 stresses high-performance transfers of data units, as well as dataflow elements with both "push" and "pull" modalities. P2 differs at its core from both PIER and Click, but subsumes many of the architectural features of both.

As in Click, nodes in a P2 dataflow graph can be chosen from a set of C++ objects called *elements*. In database systems these are often called *operators*, since they derive from logical operators in the relational algebra. Although they perform a similar function, P2 elements are typically smaller and more numerous than database operators. Unlike textbook database query plans, P2 graphs need not be trees; indeed we make heavy use of cyclic dataflow for the recursive queries that occur frequently when querying graph structures.

Elements have some number of input and output *ports*. An arc in the dataflow graph is represented by a binding between an output port on one element and an input port on another. Tuples arrive at the element on input ports, and elements emit tuples from their output ports. An input port of one element must be connected to an output port of another.

Handoff of a tuple between two P2 elements takes one of two forms, *push* or *pull*, determined when the elements are configured into a graph. In a push handoff, the source element invokes a virtual method on the destination, passing the tuple on the call stack, while in a pull handoff the destination calls the source requesting the tuple, which is returned as the result of the call. We return to the choice of connection types at the end of this section.

While P2 resembles Click in its use of push and pull elements, the implementation of dataflow elements in P2 differs from Click in significant ways, as a result of different requirements.

First, the common case in a router is that a packet traverses a single path through the dataflow graph. Consequently Click implements copy-on-write for packets that must be modified (for example, to implement multicast). This has the additional benefit of very lightweight hand-offs of packets between elements – throughput is of primary concern in Click, and inter-element handoff is simply pointer passing through a virtual function call.

In contrast, the dataflow graphs that the P2 planner generates from *NDlog* specifications have many more branching points and tuples can traverse more than one path. For example, a tuple might be stored in a table but also forwarded to another element as an event notification.

Second, P2 passes tuples, not packets. Elements in P2 implement database relational operators as well as standard packet routing functions, which means flows frequently block and unblock. In Click, a flow event is typically initiated by a packet arriving over the network, queues rarely block when full (instead, they implement an explicit drop policy as in most other routers), and consequently Click's design can process packets efficiently using only event-driven scheduling of dataflow, together with "active elements," invoked periodically by the Click scheduler.

In contrast, not only do P2 dataflow graphs tend to branch more, but tuples are frequently generated inside the dataflow graph in response to the arrival of other tuples – most commonly during equijoin operations, which are fundamental to *NDlog*'s rule constructs.

Furthermore, the consequences of dropping tuples due to queue overflow in P2 are much more undesirable than the dropping of a packet in a router under high load. Many queue elements in P2 dataflow graphs therefore "block" when full or empty, and a low-latency mechanism is required for restarting a particular dataflow when new tuples arrive or space becomes available.

P2 therefore implements a simple signaling facility to allow elements to restart flows they have previously blocked. An extra argument to each "push" or "pull" invocation between elements specifies a callback (in effect, a continuation) to be invoked at some later stage *if and only if* the dataflow has been stalled as a result of the call.

For a "pull" transition, if the pull call returns no tuple then there is no data available. When a tuple does become available, the callback previously passed with the pull is invoked. This call will typically happen as part of a push transition into the source element (e.g., in the case of equijoins) or the passage of time (e.g., in a rate limiter), and the recipient of the callback will generally schedule a deferred procedure call to retry the pull as soon as possible.

"Push" transitions operate slightly differently, since the coupling of control flow and dataflow means that the destination of a push has to accept the tuple – if it did not, any state operations that occurred previously in the dataflow chain would have to be undone. As a result, push calls are always assumed to succeed, and return a boolean indicating whether it is acceptable to call push *again*. If not, the callback will be invoked at some later stage as with pull.

The use of callbacks in this way removes from the element implementation itself any scheduling decisions, while imposing a minimum of policy. P2's transitions are not as efficient as Click's but are still very fast – most take about 50 machine instructions on an IA32 processor, or 75 if the callback is invoked.

4.3.1 Dataflow elements

This section gives a brief overview of the suite of dataflow elements implemented in P2. To start with, P2 provides the relational operators found in most database systems, as well as query processors like PIER [59]: selection, projection, streaming relational join operations such as pipelined hash-joins [128], "group-by," and various aggregation functions. Since one of our motivations in designing P2 was to investigate the applicability of the dataflow element model for distributed computing, we have tried to push as much functionality of the system as possible into dataflow elements.

One example of this is in P2's networking stack. Systems like PIER [59] abstract details of transport protocols, message formats, marshaling, etc., away from the dataflow framework, and operators only deal with fully unmarshaled tuples. In contrast, P2 explicitly uses the dataflow model to chain together separate elements responsible for socket handling, packet scheduling, congestion control, reliable transmission, data serialization, and dispatch (see [123]).

A variety of elements form a bridge between the dataflow graph and persistent state in the form of stored tables. P2 has elements that store incoming tuples in tables, lookup elements that can iteratively emit all tuples in a table matching a search filter, and aggregation elements that maintain an up-to-date aggregate (such as max, min, count, etc.) on a table and emit it whenever it changes. Tables are frequently shared between elements, though some elements generate their own private tables. For example, the element responsible for eliminating duplicate results in a dataflow uses a table to keep track of what it has seen so far. Like Click, P2 includes a collection of general-purpose "glue" elements, such as a queue, a multiplexer, a round-robin scheduler (which, when pulled, pulls tuples from its inputs in order), etc. Finally, for debugging purposes, print elements that can be inserted to "watch" tuples based on table name (specified via a special "*watch(tableName)*" statement within the *NDlog* program) entering and leaving the dataflow.

It is quite simple to add new elements to the collection provided by P2, but at present the planner is not yet designed to be easily extensible. To use a new element class, one must either "hand-wire" dataflow diagrams as in Click [65] and PIER [59], or modify the planner to translate *NDlog* into dataflows that use the new element.

4.4 Network State Storage and Management

Network state is stored in *tables*, which contain tuples with expiry times and size constraints that are declaratively specified at table creation time as described in Chapter 2. Duplicate entries (tuples) are allowed in tables, and the mechanisms for maintaining these duplicates differ based on whether they are hard-state or soft-state tables as defined in Chapter 2. In *hard-state* tables, a derivation count is maintained for each unique tuple, and each tuple is deleted when its count reaches zero. In *soft-state* tables, each unique tuple has an associated lifetime that is set based on the specified expiration of its table during creation time. Duplicates result in extension of tuple lifetime, and each tuple is deleted upon expiration based on its lifetime. We enforce the lifetimes of soft-state tuples by purging the soft-state tables of any expired tuples whenever they are accessed. Tables are named using unique IDs, and consequently can be shared between different queries and/or dataflow elements.

As basic data types, P2 uses *Value*s, and *Tuple*s. A *Value* is a reference-counted object used to pass around any scalar item in the system; *Value* types include strings, integers, timestamps, and large unique identifiers. The *Value* class, together with the rules for converting between the various value types, constitute the concrete type system of P2. A *Tuple* is a vector of *Values*, and is the basic unit of data transfer in P2. Dataflow elements, described below, pass tuples between them, and tables hold sets of tuples.

Queries over tables can be specified by filters, providing an expressivity roughly equivalent to a traditional database query over a single table. In-memory indices (implemented using standard hash tables) can be attached to attributes of tables to enable quick equality lookups. Note that the table implementation – including associated indices – is a node-local construct.

The current in-memory implementation serves our requirements for implementing the networks discussed in this dissertation, all of which tend to view their routing tables as *soft-state*. Our event-driven, run-to-completion model obviates the need for locking or transaction support in our application, and relatively simple indices suffice to meet our performance requirements. In the future, there is clearly scope for table implementations that use stable storage for persistent data placement, or that wrap an existing relational database implementation.

4.5 Summary

In this chapter, we presented an overview of the P2 declarative networking system, with an emphasis on its architecture, various components (planner, dataflow installer, dataflow engine), dataflow framework and network state management. In the next chapter, we will describe the *planner* component in greater detail, and describe how *NDlog* programs can be compiled into dataflow-based execution plans to implement the network protocols using the P2 dataflow engine.

Chapter 5

Processing NDlog Programs

One of the main challenges of using a declarative language is to ensure that the declarative specifications, when compiled and executed, result in correct and efficient implementations that are faithful to the program specifications. This is particularly challenging in a distributed context, where asynchronous messaging and the unannounced failure of participants make it hard to reason about the flow of data and events in the system as a whole. In this chapter, we address this challenge by describing the steps required for the P2 planner to automatically and correctly generate execution plans from the *NDlog* rules.

The chapter is organized as follows. In Section 5.1, we describe the steps required to generate execution plans for a centralized Datalog program in the P2 system using standard recursive query processing techniques. We then extend the centralized techniques to execute distributed *NDlog* rules in Section 5.2. Based on the distributed execution plans, we motivate, propose and prove correct in Section 5.3 pipelined query evaluation techniques that are necessary for efficiency in the distributed settings. In Section 5.4, we discuss how we can ensure correct semantics of long-running *NDlog* programs in dynamic networks using incremental view maintenance techniques. In Section 5.5, we build upon all of the above techniques for processing distributed *soft-state* rules, which can gracefully tolerate failures and lost messages.

5.1 Centralized Plan Generation

In this section, we describe the steps required to generate execution plans of a centralized Datalog program in the P2 system. We utilize the *semi-naïve* [11; 12; 15] fixpoint evaluation mechanism, which is the standard method used to evaluate Datalog programs correctly with no redundant computations. We provide a high-level overview of semi-naïve evaluation (SN), and then use the *Shortest-Path* program (Figure 2.4 in Chapter 2) as an example to demonstrate how SN is achieved in the P2 system.

5.1.1 Semi-naïve Evaluation

The first step in SN is the *semi-naïve rewrite*, where each datalog rule is rewritten to generate a number of *delta rules* to be evaluated. Consider the following rule:

$$p :- p_1, p_2, ..., p_n, b_1, b_2, ..., b_m. \tag{5.1}$$

$p_1, ..., p_n$ are *derived predicates* and $b_1, ..., b_m$ are *base predicates*. Derived predicates refer to intensional relations that are derived during rule execution. Base predicates refer to extensional (stored) relations whose values are not changed during rule execution. The *SN rewrite* generates n *delta rules*, one for each derived predicate, where the k^{th} delta rule has the form[1]:

$$\triangle p^{new} :- p_1^{old}, ..., p_{k-1}^{old}, \triangle p_k^{old}, p_{k+1}, ..., p_n, b_1, b_2, ..., b_m. \tag{5.2}$$

[1]These delta rules are logically equivalent to rules of the form $\triangle \mathrm{p}_j^{new}$:- p_1, p_2, ..., p_{k-1},$\triangle \mathrm{p}_k^{old}$,$p_{k+1}$, ..., $p_n, b_1, b_2, ..., b_m$, and have the advantage of avoiding redundant inferences within each iteration.

In each delta rule, $\triangle \mathrm{p}_k^{old}$ is the *delta predicate*, and refers to p_k tuples generated for the first time in the previous iteration. p_k^{old} refers to all p_k tuples generated before the previous iteration. For example, the following rule r2-1 is the delta rule for the recursive rule r2 from the Datalog program shown in Figure 2.1 from Chapter 2:

$$r2-1\triangle path^{new}(S,D,Z,C) : -\#link(S,Z,C1),\triangle path^{old}(Z,D,Z2,C2),C = C1+C2. \tag{5.3}$$

The only derived predicate in rule r2 is *path*, and hence, one delta rule is generated. All the delta rules generated from the rewrite are then executed in synchronous rounds (or iterations) of computation, where input tuples computed in the previous iteration of a recursive rule execution are used as input in the current iteration to compute new tuples. Any new tuples that are generated for the first time in the current iteration are then used as input to the next iteration. This is repeated until a fixpoint is achieved (*i.e.*, no new tuples are produced).

Algorithm 5.1 summarizes the basic semi-naïve evaluation used to execute these rules in the P2 system. In this algorithm, P2 maintains a buffer for each delta rule, denoted by B_k. This buffer is used to store p_k tuples generated in the previous iteration ($\triangle p_k^{old}$). Initially, p_k, p_k^{old}, $\triangle p_k^{old}$ and $\triangle p_k^{new}$ are empty. As a base case, we execute all the rules to generate the initial p_k tuples, which are inserted into the corresponding B_k buffers. Each iteration of the while loop consists of flushing all existing $\triangle p_k^{old}$ tuples from B_k and executing all the delta rules to generate $\triangle p_j^{new}$ tuples, which are used to update p_j^{old}, B_j and p_j accordingly. Note that only new p_j tuples generated in the current iteration are inserted into B_j for use in the next iteration. Fixpoint is reached when all buffers are empty.

Algorithm 5.1 Semi-naïve Evaluation in P2

```
execute all rules
foreach derived predicate $p_k$
  $B_k \leftarrow p_k$
end
while $\exists B_k.size > 0$
    $\forall B_k$ where $B_k.size > 0, \triangle p_k^{old} \leftarrow B_k.flush()$
    execute all delta rules
    foreach derived predicate $p_j$
      $p_j^{old} \leftarrow p_j^{old} \cup \triangle p_j^{old}$
      $B_j \leftarrow \triangle p_j^{new} - p_j^{old}$
      $p_j \leftarrow p_j^{old} \cup B_j$
      $\triangle p_j^{new} \leftarrow \emptyset$
    end
end
```

5.1.2 Dataflow Generation

Algorithm 5.1 requires executing the delta rules at every iteration. These delta rules are each compiled into an execution plan, which is in the form of a P2 dataflow *strand*, using the conventions of the P2 dataflow framework described in Chapter 4. Each dataflow strand implements a delta rule via a chain of relational operators. In the rest of this chapter, we refer to the dataflow strand for each delta rule as a *rule strand*.

For each delta rule, each rule strand takes as input its *delta predicate* (prepended with $\triangle$). This input is then used as input to the strand which implements a sequence of elements implementing relational equijoins. Since tables are implemented as main-memory data structures with local indices over them, tuples from the stream are pushed into an equijoin element, and all matches in the table are found via an index lookup.

After the translation of the equijoins in a rule, the planner creates elements for any selection filters, which evaluate the selection predicate over each tuple, dropping those for which the result is false. In some cases, we can optimize the dataflow to push a selection upstream of an equijoin, to limit the state and work in the equijoin, following traditional database rules on the commutativity of join and selection.

Aggregate operations like *min* or *count* are translated after equijoins and selections, since they operate on fields in the rule head. Aggregate elements generally hold internal state, and when a new tuple arrives, compute the aggregate incrementally. The final part of translating each rule is the addition of a "projection" element that constructs a tuple matching the head of the rule.

r2-1 $\triangle$path^{new}(S,D,Z,C) :- #link(S,Z,C1), $\triangle$path^{old}(Z,D,Z2,C2), C = C1 + C2.

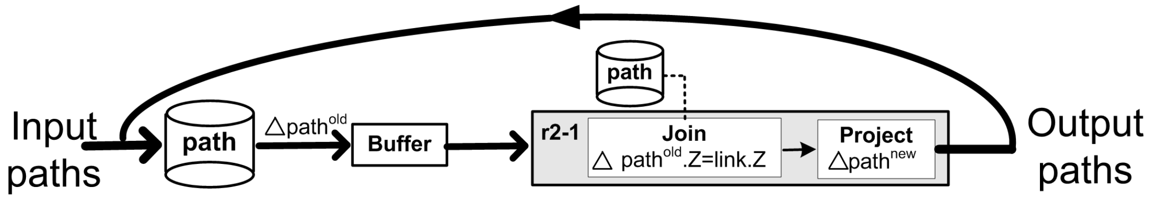

Figure 5.1: Rule strand for delta rule r2-1 in P2.

Figure 5.1 shows the dataflow realization for delta rule r2-1. We repeat the rule above the dataflow for convenience. The example rule strand receives new $\triangle path^{old}$ tuples generated in the previous iteration to generate new paths ($\triangle path^{new}$) which are then "wrapped-around" and inserted into the *path* table (with duplicate elimination) for further processing in the next iteration. In effect, semi-naïve evaluation achieves the computation of paths in synchronous rounds of increasing hop counts, where paths that have been previously in the previous round are used to generate new paths in the next iteration.

5.2 Distributed Plan Generation

In this section, we demonstrate the steps required to generate the execution plans for distributed *NDlog* rules. In Chapter 2, we introduced the concept of distributed *NDlog* rules, where the rule body predicates have different location specifiers. These distributed rules cannot be executed at a single node, since the tuples that must be joined are situated at different nodes in the network. Prior to the SN rewrite step, an additional *localization rewrite* step ensures that all body predicates for tuples to be joined are at the same node. After applying the localization rewrite to all distributed rules, all localized rules will have rule bodies that are locally computable and hence can be processed in a similar fashion as centralized Datalog rules.

```
sp2 path(@S,D,Z,P,C) :- #link(@S,Z,C1), path(@Z,D,Z2,P2,C2), C = C1 + C2,
                                        P = f_concatPath(S,P2).
```

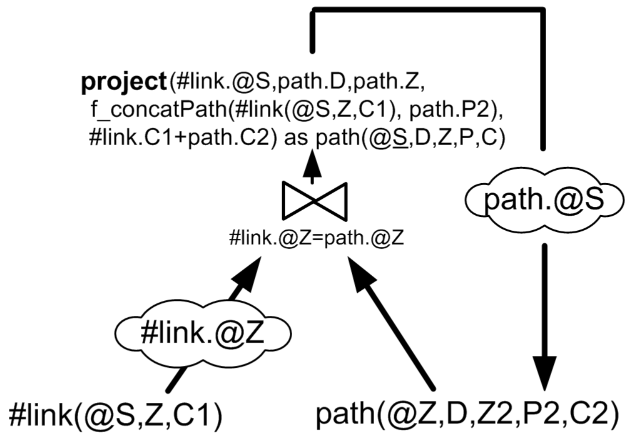
project(#link.@S,path.D,path.Z,
f_concatPath(#link(@S,Z,C1), path.P2),
#link.C1+path.C2) as path(@S,D,Z,P,C)
path.@S
#link.@Z=path.@Z
#link.@Z
#link(@S,Z,C1)
path(@Z,D,Z2,P2,C2)

Figure 5.2: Logical query plan for distributed rule sp2 shown above the figure.

5.2.1 Localization Rewrite

To provide a high-level intuition for the localization rewrite, we consider the distributed rule sp2 from the *Shortest-Path* program presented in Chapter 2. This rule is distributed because the *link* and *path* predicates in the rule body have different location specifiers, but are joined by a common “Z” field. Figure 5.2 shows the corresponding logical query plan depicting the distributed join. The clouds represent an “exchange”-like operator [51] that forwards tuples from one network node to another; clouds are labeled with the link attribute that determines the tuple’s recipient. The first cloud (*#link.@Z*) sends link tuples to the neighbor nodes indicated by their destination address fields. The second cloud (*path.@S*) transmits new *path* tuples computed from the join for further processing, setting the recipient according to the source address field.

Based on the above distributed join, rule sp2 can be rewritten into the following two rules. Note that all predicates in the body of sp2a have the same location specifiers; the same is true of sp2b. Since *linkD* is derived from the materialized table *#link*, we need to also declare *linkD* via the *materialize* statement, and set its lifetime and size parameters to be the same as that of the *#link* table.

```
materialize(linkD,infinity,infinity,1,2).
sp2a linkD(S,@Z,C) :- #link(@S,Z,C).
sp2b path(@S,D,Z,P,C) :- #link(@Z,S,C3),linkD(S,@Z,C1),path(@Z,D,Z2,P2,C2),
                          C = C1 + C2, P = f_concatPath(S,P2).
```

Figure 5.3: Localized rules for distributed rule sp2.

The rewrite is achievable because the *link* and *path* predicates, although at different locations, share a common join address field. In Algorithm 5.2, we summarize the general rewrite technique for an input set of link-restricted rules R. In the pseudocode, for simplicity, we assume that the location specifiers of all the body predicates are sorted (*@S* followed by @*D*); this can be done as a preprocessing step. The algorithm as presented here assumes that all links are bidirectional, and may add a *#link(@D,S)* to a rewritten rule to allow for backward propagation of messages. If links are not bidirectional, this means that distributed rules whose rewritten rules require the backward propagation of messages along *#link(@D,S)* cannot be localized; this can be syntactically checked at the time of parsing. From this point on we assume bidirectional links in our discussion.

Algorithm 5.2 Rule Localization Rewrite for Link-Restricted Rules

```
proc RuleLocalization(R)
  while ∃ rule r ∈ R: h(@L,...) :- #link(@S,D,...), $p_1$(@S,..),..,$p_i$(@S,...),
                                   $p_{i+1}$(@D,...),..,$p_n$(@D,..)
      R.remove(r)
      R.add(hD(S,@D,..) :- #link(@S,D,..),..,$p_i$(@S,..).)
      if @L = @D
        then R.add(h(@D,..) :- hD(S,@D,..), $p_{i+1}$(@D,..),.., $p_n$(@D,..).)
         else R.add(h(@S,..) :- #link(@D,S), hD(,@D..), $p_{i+1}$(@D,..),.., $p_n$(@D,..).)
```

Observation 5.1 *Every link-restricted* NDlog *program, when rewritten using Algorithm 5.2, produces an equivalent program where the following holds:*

1. *The body of each rule can be evaluated at a single node.*
2. *The communication required to evaluate a rule is limited to sending derived tuples over links that exists in a link relation.*

The equivalence statement in the above observation can be easily shown, by examining the simple factoring of each removed rule into two parts. The remainder of the observation can be verified syntactically in the added rules in Algorithm 5.2.

sp2a $\triangle$linkD^{new}(S,@Z,C) :- $\triangle$*#link*^{old}(@S,Z,C).

sp2b-1 $\triangle$path^{new}(@S,D,Z,P,C) :- #link(@Z,S,C3), linkD(S,@Z,C1),
$\triangle$path^{old}(@Z,D,Z2,P2,C2), C = C1 + C2,
P = *f_concatPath*(S,P2).

sp2b-2 $\triangle$path^{new}(@S,D,Z,P,C) :- #link(@Z,S,C3), $\triangle$linkD^{old}(S,@Z,C1),
path(@Z,D,Z2,P2,C2), C = C1 + C2,
P = *f_concatPath*(S,P2).

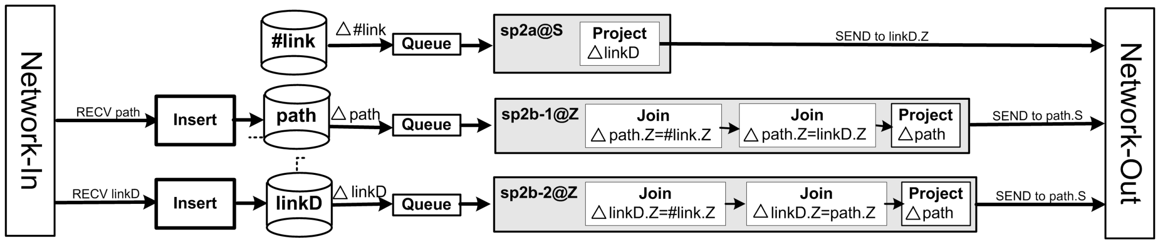

Figure 5.4: Delta rules and compiled rule strands for localized rules sp2a and sp2b.

5.2.2 Distributed Dataflow Generation

After rule localization, the SN rewrite described in Section 5.1.1 is used to generate delta rules that are compiled into rule strands. In Figure 5.4, we provide an example of the delta rules and compiled rule strands for the localized rules sp2a and sp2b shown in Figure 5.3.

In addition to creating the relational operations described in the previous section on rule strand generation, the planner also constructs the other portions of the dataflow graph in order to support distribution: the network processing elements which includes multiplexing and demultiplexing tuples, marshaling, unmarshaling and congestion control. As with Click, it also inserts explicit queue elements where there is a push/pull mismatch between two elements that need to be connected.

For simplicity, we represent the network packet processing, demultiplexing and multiplexing elements described in Section 4.2 as *Network-In* and *Network-Out* blocks in the figure, and only show the elements for the rule strands. Unlike the centralized strand in Figure 5.1, there are now three rule strands. The extra two strands (sp2a@S and sp2b-2@Z) are used as follows. Rule strand sp2a@S sends all existing links to the destination address field as *linkD* tuples. Rule strand sp2b-2@Z takes the new *linkD* tuples it received via the network, stores them using the *Insert element*. Each new *linkD* tuple (with duplicate elimination) is then used to perform a join operation with the local *path* table to generate new paths.

5.3 Relaxing Semi-naïve Evaluation

In our distributed implementation, the execution of rule strands can depend on tuples arriving via the network, and can also result in new tuples being sent over the network. Traditional SN completely evaluates all rules on a given set of facts, *i.e.*, completes the *iteration*, before considering any new facts. In a distributed execution environment where messages can be delayed or lost, the completion of an iteration in the traditional sense can only be detected by a consensus computation across multiple nodes, which is prohibitively expensive. Further, the requirement that many nodes complete the iteration together (a "barrier synchronization" in parallel computing terminology) limits parallelism significantly by restricting the rate of progress to that of the slowest node.

We address this by making the notion of iteration local to a node. New facts might be generated through local rule execution, or might be received from another node while a local iteration is in progress. We propose and prove correct two variations of SN iteration to handle this situation: *buffered SN* (BSN) and *pipelined semi-naive* (PSN). Both approaches extend SN to work in an asynchronous distributed setting, while generating the same results as SN. We further prove that these techniques avoid duplicate inferences, which would otherwise result in generating unnecessary network messages.

5.3.1 Buffered Semi-naïve Evaluation

Buffered SN (BSN) is the standard SN algorithm described in Algorithm 5.1 with the following modifications: a node can start a local SN iteration at any time its local B_k buffers are non-empty. Tuples arriving over the network while an iteration is in progress are buffered for processing in the next iteration.

By relaxing the need to run an iteration to global completion, BSN relaxes SN substantially, by allowing a tuple from a traditional SN iteration to be buffered arbitrarily, and handled in some future iteration of our choice. Consequently, BSN may generate fewer tuples per iteration, but all results will eventually be generated. We observe that since BSN uses the basic SN algorithm, BSN generates the same results as SN.

The flexibility offered by BSN on when to process a tuple could also be valuable outside the network setting, *e.g.*, a disk-based hash join could accumulate certain tuples across iterations, spill them to disk in value-based partitions, and process them in value batches, rather than in order of iteration number. Similar arguments for buffering apply to other query processing tricks: achieving locality in B-tree lookups, improving run-lengths in tournament sorts, etc.

5.3.2 Pipelined Semi-naïve Evaluation

As an alternative to BSN, *pipelined SN* (PSN) relaxes SN to the extreme of processing each tuple as it is received. This provides opportunities for additional optimizations on a per-tuple basis, at the potential cost of batch, set-oriented optimizations of local processing. New tuples that are generated from the SN rules, as well as tuples received from other nodes, are used immediately to compute tuples without waiting for the current (local) iteration to complete.

Algorithm 5.3 Pipelined SN (PSN) Evaluation.

```
execute all rules
foreach $t_k \in$ derived predicate $p_k$
  $t_k.T \leftarrow$ current_time()
  $B_k \leftarrow t_k$
end
while $\exists Q_k.size > 0$
      $t_k^{old,i} \leftarrow Q_k.dequeueTuple()$
      foreach delta rule execution
            $\triangle p_j^{new,i+1} :- p_1, p_2, ..., p_{k-1}, t_k^{old,i}, p_{k+1}, .., p_n, b_1, b_2, ..., b_m,$
             $t^{old,i}.T \geq p_1.T, t^{old,i}.T \geq p_2.T, ..., t^{old,i}.T \geq p_{k-1}.T, t^{old,i}.T \geq p_{k+1}.T, ...,$
             $t^{old,i}.T \geq p_n.T, t^{old,i}.T \geq b_1.T., t^{old,i}.T \geq b_2.T, ..., t^{old,i}.T \geq b_m.T$
          foreach $t_j^{new,i+1} \in \triangle p_j^{new,i+1}$
             if $t_j^{new,i+1} \notin p_j$
                then $p_j \leftarrow p_j \cup t_j^{new,i+1}$
                     $t_j^{new,i+1}.T \leftarrow$ current_time()
                     $Q_j.enqueueTuple(t_j^{new,i+1})$
```

Algorithm 5.3 shows the pseudocode for PSN. In PSN, the k^{th} delta rule is of the form:

$$p_j^{new,i+1} :- p_1, .., p_{k-1}, t_k^{old,i}, p_{k+1}, .., p_n, b_1, b_2, ..., b_m. \tag{5.4}$$

Each tuple, denoted t, has a superscript (*old/new*, i) where i is its corresponding iteration number in SN. Each processing step in PSN consists of dequeuing a tuple $t_k^{old,i}$ from Q_k and then using it as input into all corresponding rule strands. Each resulting $t_j^{new,i+1}$ tuple is pipelined, stored in its respective p_j table (if a copy is not already there), and enqueued into Q_j for further processing. Note that in a distributed implementation, Q_j can be a queue on another node, and the node that receives the new tuple can immediately process the tuple after the enqueue into Q_j. For example, the dataflow in Figure 5.4 is based on a distributed implementation of PSN, where incoming *path* and *linkD* tuples received via the network are stored locally, and enqueued for processing in the corresponding rule strands.

To fully pipeline evaluation, we have also removed the distinctions between p_j^{old} and p_j in the rules. Instead, a timestamp (or monotonically increasing sequence number) is added to each tuple upon its arrival (or when inserted into its table), and the join operator matches each tuple only with tuples that have the same or older timestamp. In Algorithm 5.3, we denote the timestamp of each tuple as a T field (assigned via a system call *current_time()*) and add additional selection predicates (highlighted in bold) to the k^{th} delta rule:

$$\begin{aligned} p_j^{new,i+1} :- & \; p_1,..,p_{k-1},t_k^{old,i},p_{k+1},..,p_n,b_1,b_2,...,b_m, \\ & \mathbf{t^{old,i}.T \geq p_1.T, t^{old,i}.T \geq p_2.T, ..., t^{old,i}.T \geq p_{k-1}.T, t^{old,i}.T \geq p_{k+1}.T, ...,,} \\ & \mathbf{t^{old,i}.T \geq p_n.T, t^{old,i}.T \geq b_1.T., t^{old,i}.T \geq b_2.T, ..., t^{old,i}.T \geq b_m.T.} \end{aligned}$$

Each selection predicate $t^{old,i}.T \geq p_k.T$ ensures that the timestamp of $t^{old,i}$ is greater than or equal to the timestamp of a tuple $t \in p_k$. By relaxing SN, we allow for the processing of tuples immediately upon arrival, which is natural for network message handling. The timestamp represents an alternative "book-keeping" strategy to the rewriting used in SN to ensure no repeated inferences. Note that the timestamp only needs to be assigned locally, since all the rules are localized.

While PSN enables fully pipelined evaluation, it is worth noting that PSN can allow just as much buffering as BSN with the additional flexibility of full pipelining. Consider a rule with n derived predicates and m base predicates:

$$p :- p_1, p_2, ..., p_n, b_1, b_2, ..., b_m. \tag{5.5}$$

In Appendix A.1, we prove that PSN generates the same results as SN, and does not repeat any inferences. Let $FP_S(p)$ and $FP_P(p)$ denote the result set for p for using SN and PSN respectively. We show that:

Theorem A.1: $FP_S(p) = FP_P(p)$

Theorem A.2: *There are no repeated inferences in computing $FP_P(p)$.*

In order to compute rules with aggregation (such as sp3), we utilize incremental fix-point evaluation techniques [97] that are amenable to pipelined query processing. These techniques can compute *monotonic aggregates* such as *min*, *max* and *count* incrementally based on the current aggregate and each new input tuple. Figure 5.5 shows the rule strand for rule sp3 from the *Shortest-Path* program, which computes the shortest cost (C) for any pair of source and destination paths. For each new input *path* tuple, the *Aggregate* element incrementally recomputes *spCost* tuples which are inserted into the *spCost* table.

```
sp3 spCost(@S,D,min<C>) :- path(@S,D,Z,P,C).
```

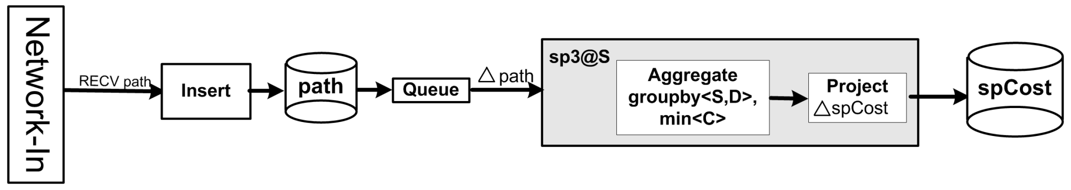

Figure 5.5: Rule strand for sp3 that computes an aggregate *spCost* over the *path* table.

5.4 Processing in a Dynamic Network

In practice, the state of the network is constantly changing during the execution of *NDlog* programs. In contrast to transactional databases, changes to network state are not isolated from *NDlog* programs while they are running. Instead, as in network protocols, *NDlog* rules are expected to perform dynamic recomputations to reflect the most current state of the network. To better understand the semantics in a dynamic network, we consider the following two degrees of dynamism:

- **Continuous Update Model:** In this model, we assume that updates occur very frequently – at a period that is shorter than the expected time for a typical program to reach a fixpoint. Hence, the query results never fully reflect the state of the network.

- **Bursty Update Model:** In this more constrained (but still fairly realistic) model, updates are allowed to happen during query processing. However, we make the assumption that after a burst of updates, the network eventually *quiesces* (does not change) for a time long enough to allow all the rule computations in the system to reach a fixpoint.

In our discussion, we focus on the bursty model, since it is amenable to analysis; our results on that model provide some intuition as to the behavior in the continuous update model. Our goal in the bursty model is to achieve a variant of the typical distributed systems notion of *eventual consistency*, customized to the particulars of *NDlog*: we wish to ensure that the eventual state of the quiescent system corresponds to what would be achieved by rerunning the rules from scratch in that state. We briefly sketch the ideas here, and follow up with details in the remainder of the section.

To ensure well-defined semantics, we use techniques from materialized view maintenance [54], and consider three types of changes:

- **Insertion:** The insertion of a new tuple at any stage of processing can be naturally handled by (pipelined) semi-naïve evaluation.

- **Deletion:** The deletion of a base tuple leads to the deletion of any tuples that were derived from that base tuple (*cascaded deletions*). Deletions are carried out incrementally via (pipelined) semi-naïve evaluation by incrementally deriving all tuples that are to be deleted.

- **Update:** An update is treated as a deletion followed by an insertion. An update to a base tuple may itself result in derivation of more updates that are propagated via (pipelined) semi-naïve evaluation.

We further allow implicit updates by primary key, where a newly generated tuple replaces an existing tuple with the same primary key (but differs on other fields). The use of pipelined SN evaluation in the discussion can be replaced with buffered SN without changing our analysis. Since some tuples in hard-state tables may have multiple derivations, we make use of the *count algorithm* [54] for keeping track of the number of derivations for each tuple, and only delete a tuple when the count is 0. We proceed to discuss these issues in detail.

5.4.1 Dataflow Generation for Incremental View Maintenance

Algorithm 5.4 Rule strands generation for incremental insertion of hard-state SN delta rules.

```
foreach $k^{th}$ delta rule $\triangle p :- p_1, p_2, ..., \triangle p_k, ..., p_n, b_1, b_2, ..., b_m$
  $RS_{ins} \leftarrow addElement(NULL, \text{Insert-Listener}(\triangle p_k))$
  foreach derived predicate $p_j$ where $j \neq k$
    $RS_{ins} \leftarrow addElement(RS_{ins}, Join(p_j))$
  end
  foreach base predicate $b_j$
    $RS_{ins} \leftarrow addElement(RS_{ins}, Join(b_j))$
  end
  $RS_{ins} \leftarrow addElement(RS_{ins}, project(\triangle p))$
  $RS_{ins} \leftarrow addElement(RS_{ins}, \text{Network-Out})$
  $RS1_{ins} \leftarrow addElement(NULL, \text{Network-In}(\triangle p))$
  $RS1_{ins} \leftarrow addElement(RS1_{ins}, Insert(\triangle p))$
end
```

Algorithms 5.4 and 5.5 shows the pseudocode for generating the rule strands for a typical delta rule of the form: $\triangle p :- p_1, p_2, ..., \triangle p_k, ..., p_n, b_1, b_2, ..., b_m$, with n derived predicates and m base predicates. The first algorithm generates rule strands RS$_{ins}$ and RS1$_{ins}$ for incremental insertions, and the second algorithm generates rule strands RS$_{del}$ and RS1$_{del}$ for for incremental deletions. In both algorithm, the function $RS \leftarrow addElement(RS, element)$ adds an element to the input rule strand RS, and then returns RS itself. For correctness, each strand has to execute completely before another strand is executed.

In Algorithm 5.4, each RS$_{ins}$ strand takes as input an *Insert-Listener($\triangle p_k$)* element that register callbacks for new insertions in the p_k table. Upon insertion of a new tuple t_k into the p_k table, the *Insert-Listener* element outputs the new tuple, which is then used to perform a series of joins with the other input tables in its rule strand to derive new p tuples. Each newly derived p tuple is then passed to a *Project($\triangle p$)*, and then sent out via the *Network-Out elements*[2]. Each RS1$_{ins}$ strand takes as input new p tuples that arrives via the network, and inserts these tuples into its local p table using the *Insert($\triangle p$)* element.

Algorithm 5.5 Rule strands generation for incremental deletion of hard-state SN delta rules.

```
foreach $k^{th}$ delta rule $\triangle p :- p_1, p_2, ..., \triangle p_k, ..., p_n, b_1, b_2, ..., b_m$
    $RS_{del} \leftarrow addElement(NULL, \text{Delete-Listener}(\triangle p_{k,del}))$
    foreach derived predicate $p_j$ where $j \neq k$
        $RS_{del} \leftarrow addElement(RS_{del}, Join(p_j))$
    end
    foreach base predicate $b_j$
        $RS_{del} \leftarrow addElement(RS_{del}, Join(b_j))$
    end
    $RS_{del} \leftarrow addElement(RS_{del}, Project(\triangle p_{del})$
    $RS_{del} \leftarrow addElement(RS_{del}, Network\_Out)$
    $RS1_{del} \leftarrow addElement(NULL, \text{Network-In}(\triangle p_{del}))$
    $RS1_{del} \leftarrow addElement(RS1_{del}, Delete(\triangle p))$
end
```

The RS$_{del}$ and RS1$_{del}$ strands in Algorithm 5.5 are generated in a similar fashion for incremental deletions. The RS$_{del}$ strand take as input tuples from a *Delete-Listener($\triangle p_{k,del}$)* element that outputs p_{del} tuples that have been deleted from the p_k table. The RS1$_{del}$ strand

[2]Note that outbound p tuples generated by RS$_{ins}$ that are destined for local consumption are "wrapped around" to the *Network-In* element as input to RS1$_{ins}$ of the same dataflow locally, as described in Section 4.2

receives these tuples, and then delete those with the same values from the local *p* table using the *Delete(△p)* element.

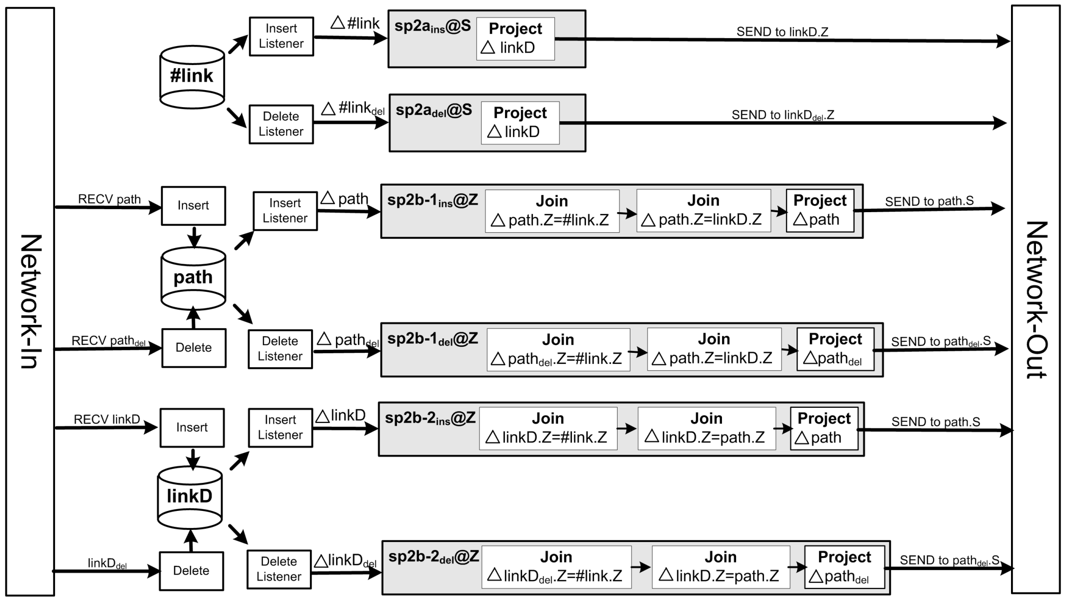

Figure 5.6: Rule strands for the SN delta rules sp2a, sp2b-1 and sp2b-2 with incremental maintenance.

To provide a concrete example, Figure 5.6 shows an example of compiled dataflow with rule strands for the delta rules sp2a, sp2b-1 and sp2b-2 that we presented earlier in Section 5.2. For each delta rule, applying Algorithms 5.4 and 5.5 result in several strands for incremental insertions and deletions. These are denoted by strand labels with subscripts *ins* and *del* respectively in Figure 5.6. For example, strands sp2a$_{ins}$@S and sp2a$_{del}$@S are generated from the delta rule sp2a, and used to implement the incremental recomputation of *linkD* table based on modifications to the *#link* table. Similarly, strands sp2b-1$_{ins}$@S and sp2b-1$_{del}$@S are generated from delta rule sp2b-1, and strands sp2b-2$_{ins}$@S and sp2b-2$_{del}$@S are generated from delta rule sp2b-2.

In handling rules with aggregates, we apply techniques for incremental computation of aggregates [97] in the presence of updates. The arrival of new tuples may invalidate existing aggregates, and incremental recomputations can be cheaper than computing the entire aggregate from scratch. For example, the re-evaluation cost for min and max aggregates are shown to be $O(log\ n)$ time and $O(n)$ space [97]. This is implemented using the *Aggregate* element shown previously in Figure 5.5. The *Aggregate* element will recompute and output any *spCost* tuples whose aggregate value has been updated as a result of updates to the underlying *path* table.

5.4.2 Centralized Execution Semantics

Before considering the distributed execution semantics of *NDlog* programs, we first provide an intuitive example for the centralized case. Figure 5.7 shows a *derivation tree* for *path(@e,d,a,[e,a,b,d],7)* based on the *Shortest-Path* program. The leaves in the tree are the *#link* base tuples. The root and the intermediate nodes are tuples recursively derived from the children inputs by applying either rules sp1 and sp2. When updates occur to the base tuples, changes are propagated up the tree to the root. The left diagram shows updating the tree due to a change in base tuple *#link(@a,b,5)*, and the right diagram shows the deletion of *#link(@b,e,1)*.

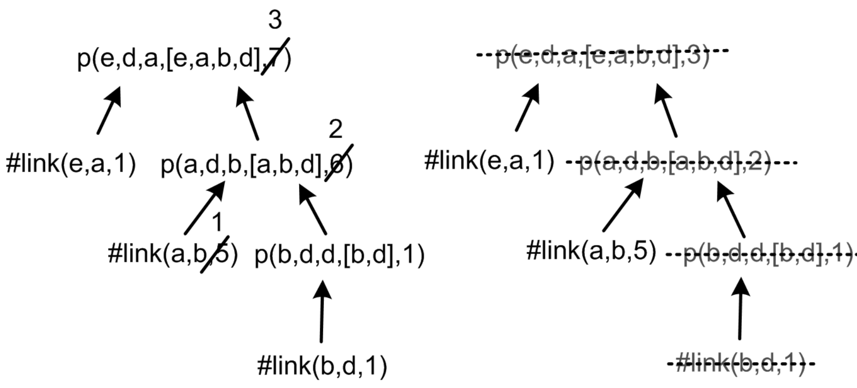

Figure 5.7: Derivation tree for derived *path* tuple from *a* to *e*.

For example, when the cost of *#link(@a,b,5)* is updated from 5 to 1, there is a deletion of *#link(@a,b,5)* followed by an insertion of *#link(@a,b,1)*. This in turn results in the deletion of *path(@a,d,b,[a,b,d],6)* and *path(@e,d,a,[e,a,b,d],7)*, followed by the derivation of *path(@a,d,b,[a,b,d],2)* and *path(@e,d,a,[e,a,b,d],3)*. Similarly, the deletion of *#link(@b,d,1)* leads to the deletion of *path(@b,d,d,[b,d],1)*, *path(@a,d,b,[a,b,d],2)*, and then *path(@e,d,a,[e,a,b,d],3)*.

Let FP_p be the set of tuples derived using PSN under the bursty model, and FFP_p be the set of tuples that would be computed by PSN if starting from the quiesced state. In Appendix A.2, we prove the following theorem:

Theorem A.3: $FP_p = FFP_p$ *in a centralized setting.*

The proof requires that all changes (inserts, deletes, updates) are applied in the same order in which they arrive. This is guaranteed by the FIFO queue of PSN and the use of timestamps.

5.4.3 Distributed Execution Semantics

In order for incremental evaluation to work in a distributed environment, it is essential that along any link in the network, there is a FIFO ordering of messages. That is, along any link literal *#link(s,d)*, facts derived at node s should arrive at node d in the same order in which they are derived (and vice versa). This guarantees that updates can be applied in order. Using the same definition of FP_p and FFP_p as before, assuming the link FIFO ordering, in Appendix A.2, we prove the following theorem:

Theorem A.4: $FP_p = FFP_p$ *in a distributed setting with FIFO links.*

5.5 Processing Soft-state Rules

Up to this point in the chapter, we have focused on the processing of *hard-state rules*. In this section, we build upon the earlier techniques to process *soft-states rules*. Recall from

Section 2.5 that a rule is considered soft-state if it contains at least one soft-state predicate in the rule head or body.

Soft-state relations are stored in *soft-state tables* within the P2 System as described in Section 4.4. Unlike hard-state tables, these tables store tuples only for their specified lifetimes and expire them in a manner consistent with traditional soft-state semantics. Timeouts can be managed lazily in soft-state tables by purging any expired soft-state tuples whenever table are accessed. Unlike hard-state tables, these soft-state tables do not require maintaining a derivation count for each unique tuple. Instead, soft-state tuples that are inserted into their respective tables will extend the lifetime of identical tuples.

Prior to applying SN rewrite, the processing of soft-state rules require the same *localization rewrite* step described in Section 5.2. After localization, the SN rewrite is applied to all soft-state rules. Consider a soft-state rule of the form:

$$p :- s_1, s_2, ...s_m, h_1, h_2, ..., h_n, b_1, b_2, ...b_o \tag{5.6}$$

where $s_1, s_2, ..., s_m$ are m soft-state derived predicates, $h_1, h_2, ..., h_n$ are hard-state derived predicates, and $b_1, b_2, ..., b_o$ are base predicates. The SN rewrite generates $m+n$ delta rules, one for each soft-state and hard-state derived predicate, where the k^{th} *soft-state delta rule* takes as input $\triangle s_k$ tuples:

$$\triangle p :- s_1, s_2, ..., \triangle s_k, ..., s_m, h_1, h_2, ..., h_n, b_1, b_2, ...b_o. \tag{5.7}$$

In addition, the j^{th} *hard-state delta rule* takes as input $\triangle h_j$ tuples:

$$\triangle p :- s_1, ..., s_k, ..., s_m, h_1, h_2, ..., \triangle h_j, ..., h_n, b_1, b_2, ..., b_o. \tag{5.8}$$

Following the generation of delta rules, the same Algorithm 5.4 is used to generate the strands for incremental insertions in a similar fashion as hard-state rules. However, instead of using Algorithm 5.5 for generating strands for incremental deletions, Algorithm 5.6 is used to generate strands for *incremental refreshes*. The difference is due to soft-state rules being incrementally maintained using *cascaded refreshes* instead of *cascaded deletions* (See Section 2.5). In Algorithm 5.4, the strand RS_{ref} takes as input a *Refresh-Listener($\triangle p_{k,ref}$)* element that outputs soft-state p_k tuples that have been refreshed. These p_k tuples are then used to derive p tuples, which are then inserted by the RS1_{ref} into local p tables. If p is a soft-state relation, these new insertions will lead to further refreshes being generated, hence achieving cascaded refreshes.

Algorithm 5.6 Rule Strands generation for incremental refresh of soft-state delta rules.

```
foreach delta rule $\triangle p :- p_1, p_2, ..., \triangle p_k, ..., p_n, b_1, b_2, ..., b_m$
  $RS_{ref} \leftarrow addElement(NULL, \text{Refresh-Listener}(\triangle p_{k,ref}))$
  foreach derived predicate $p_j$ where $j \neq k$
    $RS_{ref} \leftarrow addElement(RS_{ref}, Join(p_j))$
  end
  foreach base predicate $b_j$
    $RS_{ref} \leftarrow addElement(RS_{ref}, Join(b_j))$
  end
  $RS_{ref} \leftarrow addElement(RS_{ref}, Project(\triangle p)$
  if $(p.loc = p_k.loc)$
    then $RS_{ins} \leftarrow addElement(RS_{ins}, Insert(\triangle p))$
    else
      $RS_{ref} \leftarrow addElement(RS_{ref}, Network\_Out)$
      $RS1_{ref} \leftarrow addElement(NULL, \text{Network-In}(\triangle p))$
      $RS1_{ref} \leftarrow addElement(RS1_{ref}, Insert(\triangle p))$
    end
```

For completeness, Figure 5.8 shows an example dataflow for a soft-state version of rule sp2, assuming that *#link* and *path* have been declared as soft-state relations. In contrast to Figure 5.6, *Refresh-Listener* elements are used instead of *Delete-Listener* elements to generate soft-state refreshes.

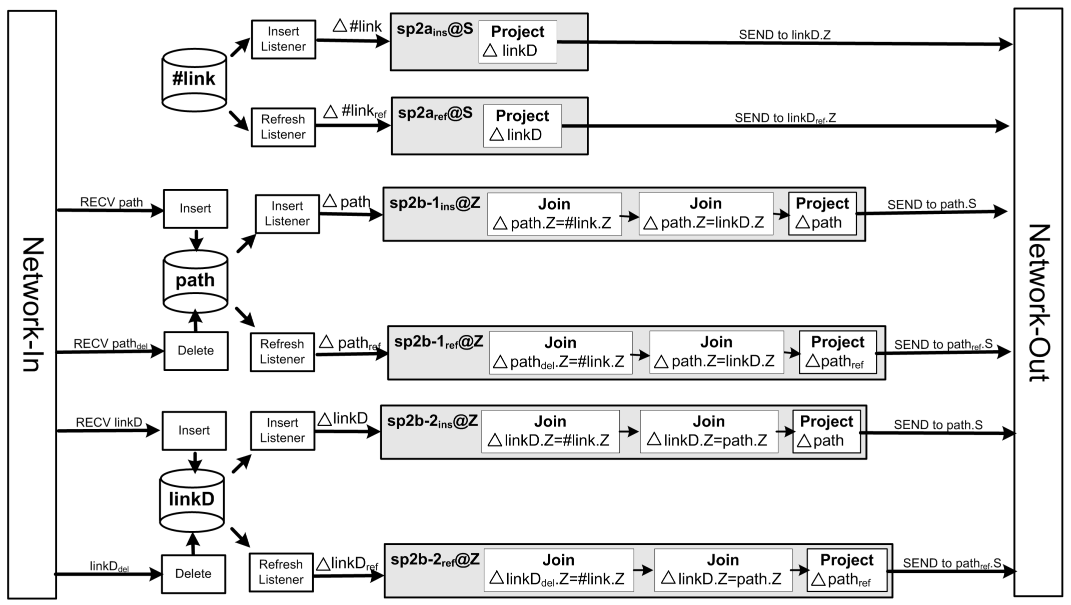
Network-In
Network-Out
#link
path
linkD
Insert Listener
Refresh Listener
Insert
Delete
RECV path
RECV path_del
RECV linkD
linkD_del
sp2a_ins@S Project △ linkD
sp2a_ref@S Project △ linkD
SEND to linkD.Z
SEND to linkD_ref.Z
sp2b-1_ins@Z Join △ path.Z=#link.Z Join △ path.Z=linkD.Z Project △path
sp2b-1_ref@Z Join △ path_del.Z=#link.Z Join △ path.Z=linkD.Z Project △path_ref
sp2b-2_ins@Z Join △ linkD.Z=#link.Z Join △ linkD.Z=path.Z Project △path
sp2b-2_ref@Z Join △ linkD_del.Z=#link.Z Join △ linkD.Z=path.Z Project △path_ref
SEND to path.S
SEND to path_ref.S

Figure 5.8: Rule strands for distributed soft-state management of delta rules sp2a, sp2b-1 and sb2b-2.

5.5.1 Event Soft-state Rules

Having presented the general steps required to process soft-state rules, in this section, we focus on a special-case soft-state rule: the *event soft-state rule* presented in Section 2.5. As a quick recap, an event soft-state rule is of the form:

$$p : -e, p_1, p_2, ..., p_n, b_1, b_2, ..., b_m. \tag{5.9}$$

The rule body consists of one event predicate e; the other predicates$p, p_1, p_2, ..., p_n$ can either soft or hard-state predicates, and $b_1, b_2, ..., b_m$ are base predicates as before.

The dataflow generation for event soft-state rules is simplified due to the fact that events are not materialized. As we discussed in Section 2.5.2, *NDlog*'s event model does not permit two events to coincide in time. Hence a rule with more than one event table would never produce any output. The only delta rule that generates any output tuples is the *event* *delta rule* that takes as input new *e* event tuples of the form:

$$\triangle p :- \triangle e, p_1, p_2, ..., p_n, b_1, b_2, b_m. \quad (5.10)$$

Since the delta predicate (prepended with $\triangle$) is essentially a stream of update events, all other delta rules do not generate any output and we can exclude them from dataflow generation.

Algorithm 5.7 Rule Strands generation for event delta rules.

```
foreach delta rule $\triangle p :- \triangle e, p_1, p_2, ..., p_n, b_1, b_2, b_m$
   /* Strand Generation */
   $RS \leftarrow addElement(NULL, \text{Network-In}(\triangle e))$
   foreach derived predicate $p_j$
      $RS \leftarrow addElement(RS_{ins}, Join(p_j))$
   end
   foreach base predicate $b_j$
      $RS \leftarrow addElement(RS, Join(b_j))$
   end
   $RS \leftarrow addElement(RS, project(\triangle p))$
   $RS \leftarrow addElement(RS, \text{Network-Out})$
   if $lifetime(p) > 0$
      $RS1 \leftarrow addElement(NULL, \text{Network-In}(\triangle p))$
      $RS1 \leftarrow addElement(RS1, Insert(\triangle p))$
   end
end
```

Algorithm 5.7 shows the pseudocode for compiling an event delta rule into its rule strands. The first strand *RS* takes each event tuple *e* that arrives over the network as input into the strand. After performing joins with all the derived predicates ($p_1, p_2, ..., p_n$) and base predicates ($b_1, b_2, ...b_m$), the computed *p* tuples are projected and sent over the network. If *p* is a materialized table (lifetime > 0), the second strand RS1 receives *p* tuples via the network, and then inserted them into the local *p* table using the *Insert($\triangle p$)* element.

As a concrete example, Figure 5.9 shows the execution plan for rules pp1 and pp2 from

```
pp1 ping(@S,D,E) :- periodic(@S,E,5), #link(@S,D).
pp2 pingMsg(S,@D,E) :- ping(@S,D,E), #link(@S,D).
```

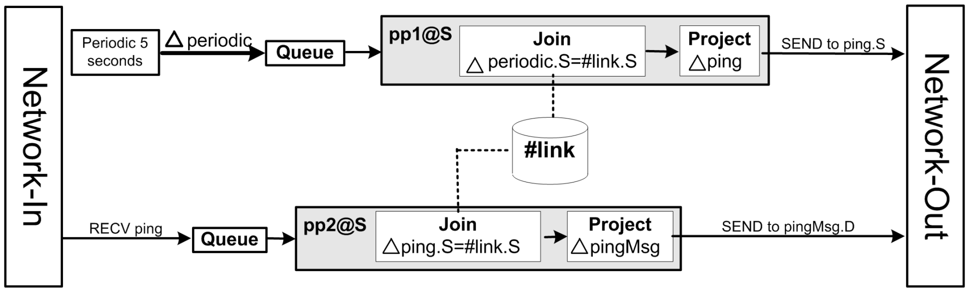

Figure 5.9: Rule strands for event soft-state rules pp1 and pp2.

the *Ping-Pong* program from Chapter 2 The first strand pp1@S takes as input a *Periodic* element that generates a *periodic(@S,E,5)* tuple every 5 seconds at node S with random event identifier *E*. This tuple is then used to join with #link tuples to generate a *ping* event tuple that is then used in strand pp2@S to generate *pingMsg* event tuples.

The output of event soft-state rules can also be an aggregate computation, which is done on a *per-event* basis. Examples of such aggregate computations are shown in rules l2 and l3 from the declarative Chord specifications in AppendixB.1. These rules computes aggregate *min* values stored in *bestLookupDist* and *lookup* tuples respectively, one for each input event.

Figure 5.10 shows the strand l2@NI generated for rule l2. This strand takes as input new *lookup* event tuples, which are then executed within the strand by joining with the *node* and *finger* tables to generate a set of matching output tuples. These output tuples are then used by the *Aggregate* element to compute a *bestLookupDist* tuple that stores the computed *min* value. Note that in this case, we have additional runtime checks in place in the dataflow execution to ensure that each *lookup* tuple is executed in its entirety within the strand to generate the *bestLookupDist* tuple before the strand processes the next *lookup* tuple.

```
l2 bestLookupDist(@NI,K,R,E,min<D>) :- nodeID(@NI,N),
                              lookup(@NI,K,R,E), finger(@NI,I,B,BI),
                              D = K - B - 1, B in (N,K).
```

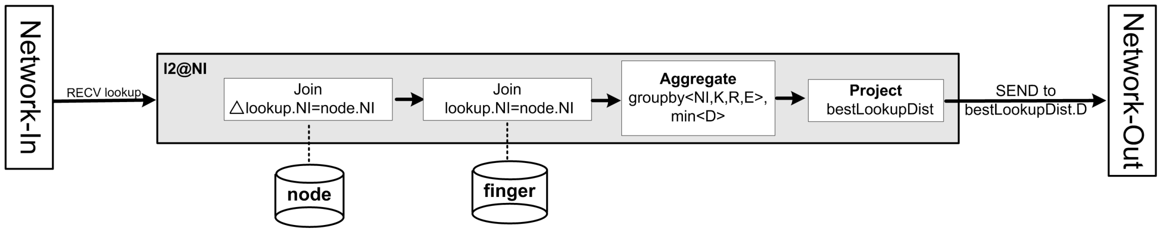

Figure 5.10: Rule strand for rule l2, an event soft-state rule with aggregation.

5.5.2 Distributed Soft State Semantics

Having described the processing of soft-state rules, we examine the distributed execution semantics of soft-state rules in dynamic networks. We consider the three types of soft-state rules defined in Section 2.5: *pure soft-state rules*, *derived soft-state rules* and *archival soft-state rules*.

In Appendix A.3, we prove that the eventual consistency semantics described in Section 5.4 can be achieved for pure soft-state rules and derived soft-state rules. Consider a pure soft-state rule of the form $s : -s_1, s_2, ...s_m,\ h_1, h_2, ...h_n$ where there are m soft-state predicates and n hard-state predicates. The rule derives a soft-state predicate s. In order for each derived s tuple to have a stable derivation at the eventual state, we require the condition that the lifetime of s exceeds the lifetime of all input soft-state relations $s_1, s_2, ..., s_m$. This ensures that all derived s tuples will not time out in between the refreshes of the soft-state inputs. This condition can be done via syntactic checks to ensure the lifetime of the derived soft-state head exceeds the lifetime of all the soft-state body predicates.

However, eventual consistency is not achievable for archival soft-state rules of the form $h : -s_1, s_2, ...s_m,\ h_1, h_2, ...h_n$, where there are m soft-state predicates, n hard-state predicates and a hard-state rule head. This is because the derived hard-state *h* tuples are stored even after the soft-state inputs have expired. In order to guarantee eventual consistency semantics, archival soft-state rules should be treated as separate from the rest of the *NDlog* program and use strictly for archival purposes only. Derivations from these rules should not be used as input to any other rules. This additional constraint on the use of archival soft-state rules can also be enforced via syntactic checks. Interestingly, if cascaded deletions are allowed for soft-state rules, eventual consistency can be achieved for these archival rules, at the expense of losing the "history" (archived data). We leave the exploration of cascaded deletions in soft-state rules as future work.

5.6 Summary

In this chapter, we described how *NDlog* programs can be processed by generating distributed dataflows. We first demonstrated how traditional semi-naïve evaluation for centralized Datalog programs can be realized in our system, and further extend the techniques to handle distributed and soft-state *NDlog* rules. We further showed how we can ensure correct semantics of long-running *NDlog* programs in dynamic networks for both hard-state and soft-state rules. In the next chapter, we present the use of *NDlog* to express more complex overlay networks.

Chapter 6

Declarative Overlays

In Chapter 3, we demonstrated the flexibility and compactness of *NDlog* for specifying a variety of routing protocols. In practice, most distributed systems are much more complex than simple routing protocols; in addition to routing, they typically also perform application-level message forwarding and handle the formation and maintenance of a network as well.

All large-scale distributed systems inherently use one or more application-level overlay networks as part of their operation. In some cases, the overlay is prominent: for example, file-sharing networks maintain neighbor tables to route queries. In other systems, the overlay or overlays may not be as explicit: for example, Microsoft Exchange email servers within an enterprise maintain an overlay network among themselves using a link-state algorithm over TCP for routing mail and status messages.

In this chapter on *declarative overlays*, we demonstrate the use of *NDlog* to implement practical application-level overlay networks. In declarative overlays, applications submit to P2 a concise *NDlog* program which describes an overlay network, and the P2 system executes the program to maintain routing tables, perform neighbor discovery and provide forwarding for the overlay.

The rest of the chapter is organized as follows. In Section 6.1, we present the execution model of declarative overlays. We then present two example *NDlog* programs: the Narada [30] mesh for end-system multicast in Section 6.2, and the Chord [114] distributed hash table in Section 6.3 respectively.

6.1 Execution Model

A typical overlay network consists of three functionalities:

- **Routing** involves the computation and maintenance of routing tables at each node based on input neighbor tables. This functionality is typically known as the *control plane* of a network.
- **Forwarding** involves the delivery of overlay messages along the computed routes based on the destination addresses of the messages. This functionality is typically known as the *forwarding* plane of a network.
- **Overlay formation and maintenance** involves the process of joining an overlay network and maintaining the neighbor set at each node. The selected neighbors are used as input to the control plane for route computations.

In declarative routing presented in Chapter 3, *NDlog* programs are used solely for programming the control plane. Hence, all our routing examples consist of *NDlog* rules that compute routes based on input links. On the other hand, in declarative overlays, *NDlog* programs implement the additional functionalities of *forwarding* and *overlay formation and maintenance*. As we see from our examples later in this chapter, these programs are more complex due to the handling of message delivery, acknowledgments, failure detection and timeouts required by the additional functionalities. Not surprisingly, the programs presented in this section utilize soft-state data and soft-state rules introduced in Chapter 2 extensively. Despite the increased complexity, we demonstrate that our *NDlog* programs are significantly more compact compared to equivalent C++ implementations.

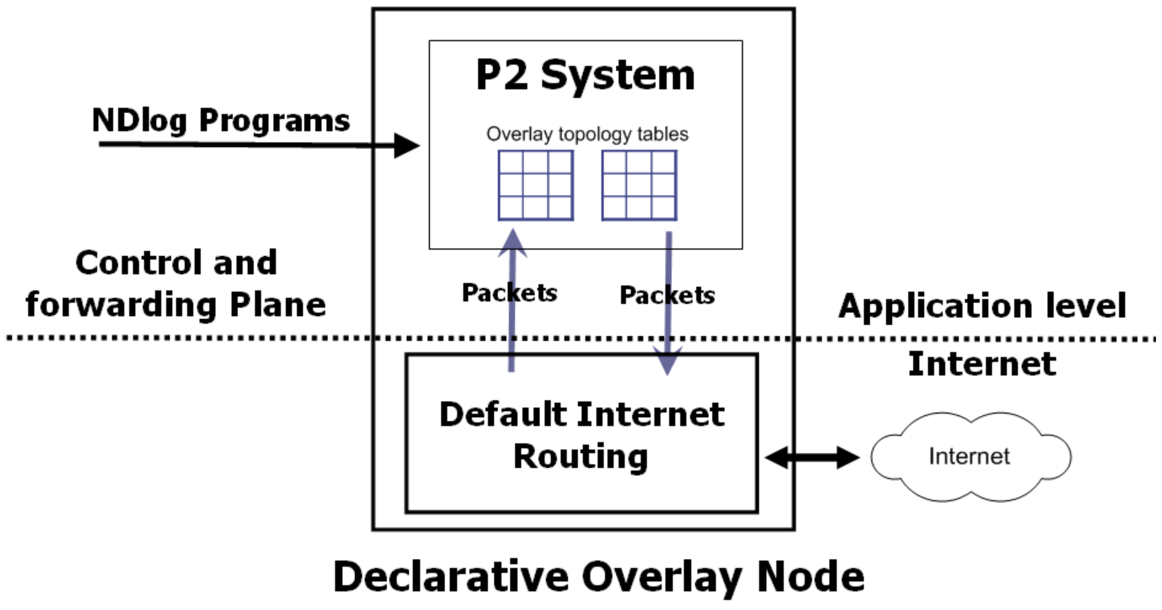

Figure 6.1: A Declarative Overlay Node.

Figure 6.1 illustrates the execution model of declarative overlays. The P2 system resides at the application level, and all messages are routed via the default Internet routing. In addition, by using the default Internet for routing between overlay nodes at the application level, we assume that there is full connectivity in the underlying network. Every node participating in the overlay network can send a message to another node via the underlying network, and there is an entry in the #link table for every source and destination pair of nodes. As a syntactic simplification, we do not use link-restricted rules in our examples below. In practice, this simplification could be supported by an *NDlog* precompiler, which allows a programmer to declare that a fully-connected topology exists, and have the parser turn off checks for link-restriction.

6.2 Narada Mesh

To provide a simple but concrete example of a declarative overlay, we first present a popular overlay network for End System Multicast (ESM) called Narada [30]. A typical ESM overlay consists of two layers: the first layer constructs and maintains a mesh connecting all members in the group, while the second layer constructs delivery trees on top of the mesh using typical multicast algorithms such as the distance vector multicast protocol (DVMRP) [36] (see Sections 3.3.2 and 3.3.6 for examples on DVMRP). In this section, we focus on the first layer: constructing a Narada-like mesh here as an example of the use of *NDlog*.

Briefly, the mesh maintenance algorithm works as follows. Each node maintains a set of neighbors, and the set of all members in the group. Every member epidemically propagates keep-alive messages for itself, associated with a monotonically increasing sequence number. At the same time, neighbors exchange information about membership liveness and sequence numbers, ensuring that every member will eventually learn of all the other group members' liveness. If a member fails to hear from a direct neighbor for a period, it declares its neighbor dead, updating its own membership state and propagating this information to the rest of the population.

In addition, each node periodically probes a random group member to measuring the round-trip latency. Based on the measured round-trip latencies to all group members, each node selects a subset of the members to be its neighbors so that its predefined utility function is maximized. In the rest of this section, we show how the mesh maintenance portion of Narada can be expressed in *NDlog*. We begin with the following table definitions and initialization rules:

```
materialize(sequence, infinity, 1, keys(2)).
materialize(neighbor, infinity, infinity, keys(2)).
materialize(member, 120, infinity, keys(2)).
e1 neighbor(@X,Y) :- periodic(@X,E,0,1), env(@X,H,Y), H = "neighbor".
e2 member(@X,A,S,T,L) :- periodic(@X,E,0,1), T = f_now(), S = 0, L = 1, A = X.
e3 member(@X,Y,S,T,L) :- periodic(@X,E,0,1), neighbor(@X,Y), T = f_now(),
                         S = 0, L = 1.
e4 sequence(@X,Sequence) :- periodic(@X,E,0,1), Sequence = 0.
```

Figure 6.2: Narada materialized tables and initialization rules

The materialized tables *neighbor* and *member* are soft-state relations with lifetime of 120 seconds, and have unbounded size. The *sequence* table is a hard-state table with unbounded size. Though not explicitly specified in the *materialize* statements, the *neighbor* contains tuples of the form *neighbor(MyAddr, NeighborAddr)* and the *member* table contains tuples of the form member(MyAddr, MemberAddr, MemberS, MemberInsertionTime, MemberLive). *MemberLive* is a boolean indicating whether the local node believes a member is alive or has failed.

rule e1 initializes the neighbor table at each node based on its local *env* table which contains its initial set of neighbors that have been preloaded into the table when the node is started. Rules *e2-4* are used to initialize the *member* table and *sequence* tables respectively. As described in Chapter 2, *periodic(@X,E,T,K)* is a built-in event predicate that is used to generate a stream of *periodic* tuples at node *X* with random event identifier *E* every *T* seconds for up to *K* tuples. Hence, the initialization rules e1 and *e2* are only invoked once.

Rule e2-3 initialize the *member* table at each node to itself and its initial set of neighbors. The *sequence(@X,Seq)* is a hard-state relation of size 1, which stores a single tuple that keeps track of the current sequence number *Seq* used in the gossip protocol.

6.2.1 Membership List Maintenance

```
r1 refreshEvent(@X) :- periodic(@X,E,5).
r2 refreshSeq@X(X,NewS) :- refreshEvent@X(X), sequence@X(X,S), NewS = S + 1.
r3 sequence@X(X,NewS) :- refreshSeq@X(X,NewS).
r4 refreshMsg(@Y,X,NewS,Addr,AS,ALive) :- refreshSeq(@X,NewS),
                        member(@X,Addr,AS,Time,ALive),
                        neighbor(@X,Y).
r5 membersCount(@X,Addr,AS,ALive,count<*>) :-
                        refreshMsg(@X,Y,YS,Addr,AS,ALive),
                        member(@X,Addr,MyS,MyTime,MyLive), X != Addr.
r6 member(@X,Addr,AS,T,ALive) :- membersCount(@X,Addr,AS,ALive,C),
                        C = 0, T = f_now().
r7 member(@X,Addr,AS,T,ALive) :- membersCount(@X,Addr,AS,ALive,C),
                        member(@X,Addr,MyS,MyT,MyLive),
                        T = f_now(), C > 0, MyS < AS.
r8 neighbor(@X,Y) :- refresh(@X,Y,YS,A,AS,L).
```

Figure 6.3: Narada Membership List Maintenance

We assume that at the start, each node begins with an initial neighbor set. Narada then periodically gossips with neighbors to refresh membership information. In Figure 6.3, the rules r1-r9 specify the rules for the periodic maintenance of the membership lists.

Rule r1 generates a *requestEvent* tuple every 3 seconds at node *X*. The request interval is set by the programmer and is used to determine the rate at which nodes in the Narada exchange membership lists.

Before a Narada node can refresh its neighbors' membership lists, it must update its own sequence number, stored in the *sequence* table. Upon generating a *refreshEvent*, rule r2 creates a new refresh sequence number *NewS* for *X* by incrementing the currently stored sequence number *NewS* in the *sequence* table. Rule r3 updates the stored sequence number. Because *sequence* is a materialized table, whenever a new *sequence* tuple is produced, as is done with rule r3, it is implicitly inserted into the associated table. Since the primary key is the sequence number itself, this new *sequence* tuple replaces the existing tuple based on our update semantics defined in Chapter 2.

In rule r4, the *refreshSeq(@X,NewS)* that is generated is then used to generate a *refresh* message tuple that is sent to each of *X*'s neighbors. Each *refresh* message tuple contains information about a membership entry as well as the current sequence number *NewS*.

Upon receiving the *refresh* message, rule r5 checks to see if the member *Addr* reported in the *refresh* message exists in the membership list. If such a member does not exist, the new member is inserted into the membership table (rule r6). If the member already exists, it is inserted into the membership table only if the sequence number in the *refresh* message is larger than that of the existing sequence number in the membership list (rule r7). The function *f_now()* is used to timestamp each *member* tuple stored.

To join the mesh, a new node need only know one member of the mesh, placing that member into its *neighbor* table. Rule r8 ensures that whenever a node receives a *refresh* message from its neighbor, it adds the sender to its neighbor set. This ensures that neighbor relationships are mutual.

6.2.2 Neighbor Selection

There are two aspects of neighbor selection in Narada: first, evicting neighbors that are no longer responding to heartbeats, and second, to select neighbors that meet certain user-defined criteria.

Figure 6.3 shows the rules l1-l4 that can be used to check neighbor liveness. Every second, rule *l1* initiates a neighbor check by which rule l2 declares *dead* a neighboring member that has failed to refresh for longer than 20 seconds. Dead neighbors are deleted from the *neighbor* table by rule l3 and rule l4 sets a dead neighbor's member entry to be "dead" and further propagated to the rest of the mesh during refreshes.

```
l1 neighborProbe(@X) :- periodic(@X,E,1).
l2 deadNeighbor(@X,Y) :- neighborProbe(@X), T = f_now(),
                         neighbor(@X,Y), member(@X,Y,YS,YT,L), T - YT > 20.
l3 delete neighbor(@X,Y) :- deadNeighbor(@X,Y).
l4 member(@X,Neighbor,DeadSequence,T,Live) :- deadNeighbor(@X,Neighbor),
                         member(@X,Neighbor,S,T1,L), Live = 0,
                         DeadSequence = S + 1, T = f_now().
```

Figure 6.4: Rules for neighbor liveness checks.

Figure 6.5 shows the rules (*n0-n3*) for probing neighbors for latency measurements. Every 2 seconds, rule n0 picks a member at random with which to measure round-trip latency. Specifically, it associates a random number with each known member, and then chooses the member associated with the maximum random number. Recall that *aggregate¡fields¿* denotes an aggregation function, *max* in this example. When a tuple appears in data stream *pingEvent*, rule n1 pings the randomly chosen member stored in the event, rule n2 echoes that ping, and rule *n3* computes the round-trip latency of the exchange.

```
n0 pingEvent(@X,Y,E,max<R>) :- periodic(@X,E,2), member(@X,Y,U,V,Z),
                              R = f_rand().
n1 ping(@Y,X,E,T) :- pingEvent(@X,Y,E,MR), T = f_now().
n2 pong(@X,Y,E,T) :- ping(@Y,X,E,T).
n3 latency(@X,Y,T) :- pong@X(X,Y,E,T1), T = f_now() - T1.
n4 ugain(@X,Z,sum<UGain>) :- latency(@X,Z,T), bestPathHop(@Z,Y,W,C),
                              bestPathHop(@X,Y,Z,UCurr), UNew = T + C,
                              UNew < UCurr, UGain = (UCurr - UNew) / UCurr.
n5 neighbor(@X,Z) :- ugain(@X,Z,UGain), UGain > addThresh.
```

Figure 6.5: Rules for neighbor selection based on latency.

Nodes use such latency measurements – along with the paths computed by a routing protocol operating on top of the mesh – to compute a utility function. A node may choose a new member to add to its current neighbor set, if adding the new member increases its utility gain above an *addition threshold*. Similarly, if the cost of maintaining a current neighbor is greater than a *removal threshold*, the node may break its link with that neighbor.

In the rules n4 and *n5* shown in Figure 6.5, we show how neighbor addition would work in an *NDlog* implementation of Narada. We assume that each node maintains a routing table over the mesh which contains for each member the next hop to that member and the cost of the resulting path; e.g., *bestPathHop(@S,D,Z,C)* indicates that node *S* must route via next-hop node *Z* to get to destination *D* with a path latency of *C*. This *bestPathHop* table can be computed by running the *distance-vector* protocol described in Section 3.3, taking as input the *neighbor* table as the input topology.

Rule n4 measures the utility gain that could be obtained if node *Z* were to become *X*'s immediate neighbor, as per the Narada definition [30]. For an individual destination *Y*, this is computed by taking the latency of *Z*'s path to *Y* and adding the latency between *X* and

Z to it. If this new path latency (assuming *Z* becomes the next hop from *X*) is lower than the current latency of *X*'s route to *Y*, then the relative decrease in latency contributes to the utility gain by adding neighbor *Z*. If this utility gain is above a threshold *addThresh*, then rule *n5* adds this new neighbor

6.3 Chord Distributed Hash Table

In this section, we present *P2-Chord*, which is a full-fledged implementation of the Chord distributed hash table [114] implemented in 48 *NDlog* rules. The entire P2-Chord specification is shown uninterrupted by discussion in Appendix B.2.

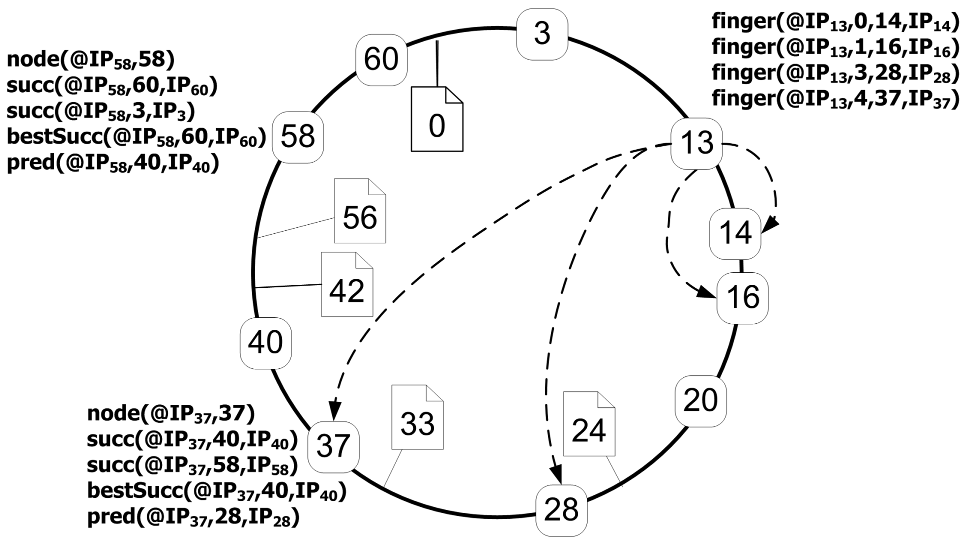

Figure 6.6: A Chord ring with the network state for node *58* and *37*, the finger entries for node *13*, and stored objects *0*, *24*, *33*, *42* and *56*. The dotted lines denote the fingers for node *13*.

Chord is essentially a mechanism for maintaining a ring-based network and routing efficiently on it. Figure 6.6 shows an example of a Chord ring. Each node in the Chord ring has a unique 160-bit node identifier. For simplicity in the figure, we show them as integers ranging from 0 to 60. Each Chord node is responsible for storing objects within a range of key-space. This is done by assigning each object with key K to the first node whose identifier is equal to or follows K in the identifier space. This node is called the *successor* of the key K. Note that data items and nodes are mapped into the same identifier space. Therefore each node also has a successor: the node with the next-higher identifier. For example, the objects with key 42 and 56 are served by node 58.

In Chord, each node maintains the IP addresses of multiple successors to form a ring of nodes that is resilient to failure. Once a node has joined the Chord ring, it maintains network state for S successors in the ring (the *succ* table) with the closest identifier distance to the node, and a single predecessor (the *pred* table of size 1) that stores the address of the node whose identifier just precedes the node. The *bestSucc* stores the address of the successor whose identifier is the closest among all the successors to the current node. For example, if $S = 2$, the successors of node 58 in Figure 6.6 are 60 and 3, its best successor is 60 and its predecessor is 40.

In order to perform scalable lookups, each Chord node also holds a *finger* table, pointing at peers whose identifier distances exponentially increase by powers of two from itself. The entries in the *finger* table are used for efficiently routing lookup requests for specific keys. There are typically 160 finger entries at each Chord node with identifier N, where the i^{th} entry stores the node that is responsible for the key $2^i + N$. In our example Chord ring, node 13 has finger entries to nodes 14, 16, 28 and 37, as denoted by the dotted lines.

6.3.1 Chord Network State

Figure 6.7 shows the materialized tables that are used to store the network state of P2-Chord. For convenience, we also show the corresponding schemas of the tables with their abbreviations are shown in Table 6.1.

Each node stores a single *landmark* tuple denoting the address of the node that it uses

```
materialize(nodeID, infinity, 1, keys(1)).
materialize(landmark, infinity, 1, keys(1)).
materialize(finger, 180, 160, keys(2)).
materialize(uniqueFinger, 180, 160, keys(2)).
materialize(bestSucc, 180, 1, keys(1)).
materialize(succ, 30, 16, keys(2)).
materialize(pred, infinity, 1, keys(1)).
materialize(join, 10, 5, keys(1)).
materialize(pendingPing, 10, infinity, keys(3)).
materialize(fFix, 180, 160, keys(2)).
materialize(nextFingerFix, 180, 1, keys(1)).
```

Figure 6.7: Materialized tables for P2-Chord.

to join the Chord network. It also stores a *nodeID* tuple that contains its node identifier. In addition, each node stores the network state for Chord in the *succ*, *pred*, *bestSucc* and *finger* tables. To illustrate, Figure 6.6 shows the network state stored at node *58* that consists of the following tuples:

- A node(@IP_{58},58) tuple, where IP_{58} denotes the IP address of node 58, and 58 is the actual identifier itself
- succ(@IP_{58},60,IP_{60}) and succ(@IP_{58},3,IP_3) tuples storing the immediate identifier and IP addresses of the two successors of node 58.
- bestSucc(@IP_{58},60,IP_{60}) and pred(@IP_{58},40,IP_{40}) tuples storing the identifier and IP addresses of the best successor and predecessor of node 58.

The figure also shows similar network state for node 37, and the four finger entries for node 13: finger(@IP_{13},0,14,IP_{14}), finger(@IP_{13},1,16,IP_{16}), finger(@IP_{13},3,28,IP_{28}) and

Predicate	**Schema**
nodeID(@NI,N)	nodeID(@NodeIP,NodeID)
landmark(@NI,N)	landmark(@NodeIP,NodeID)
finger(@NI,I,BI,B)	finger(@NodeIP,EntryNumber,BestFingerIP, BestFingerID)
uniqueFinger(@NI,I,BI,B)	uniqueFinger(@NodeIP,FingerIP)
bestSucc(@NI,N)	bestSuccessor(@NodeIP,NodeID)
succ(@NI,N)	successor(@NodeIP,NodeID)
pred(@NI,N)	predecessor(@NodeIP,NodeID)
join(@NI,E)	join(@NodeIP,EventID)
pendingPing(@NI,PI,E,T)	pendingPing(@nodeIP,PingNodeID,EventID, PingTime)
lookup(@NI,K,R,E)	lookup(@currentNodeIP,Key,RequestingNode, EventID)
lookupResults(@NI,K,R,RI,E)	lookupResults(@RequestingNodeIP,Key,ResultKey, ResultNodeIP,EventID)

Table 6.1: Predicates and corresponding schemas of materialized tables and lookup events used in P2-Chord

finger(@IP_{13},4,37,IP_{37}). Since there can be multiple finger entries pointing to the same node, the *uniqueFinger* table is used to keep track of only the unique nodes that are pointed by the finger entries.

In addition, there are other materialized tables such as *join*, *pendingPing*, *fFix* and *nextFingerFix* that are used to store intermediate state in our P2-Chord implementation. In the rest of the section, we demonstrate how different aspects of Chord can be specified in *NDlog*: *joining the Chord network*, *ring maintenance*, *finger maintenance and routing*, and *failure detection*.

```
i1 pred(@NI,P,PI) :- periodic(@NI,E,0,1), P = “NIL”, PI = ”NIL”.
i2 nextFingerFix(@NI, 0) :- periodic(@NI,E,0,1).
i3 node(@NI,N) :- periodic(@NI,E,0,1), env(@NI,H,N), H = ”node”.
i4 landmark(@NI,LI) :- periodic(@NI,E,0,1), env(@NI,H,LI), H = ”landmark”.
```

Figure 6.8: Rules for initializing a Chord node.

6.3.2 Joining the Chord Network

When a node is started, rules i1-i4 from Figure 6.8 can immediately deduce facts that set the initial state of the node. Rule i1 sets the *pred* to point to NIL indicating that there are no predecessors. Rule i2 initializes the *nextFingerFix* to be *0* for use in finger maintenance, as described in Section 6.3.4. Rule i3 initializes a *landmark(@NI,LI)* tuple in the *landmark* table of each node *NI* storing the address of the landmark node *LI*. This address is input to the P2 system via a preloaded local *env* table. The landmark *LI* is set to NIL if the node itself is the landmark. Each node also stores a *node(@NI,N)* tuple that contains the random node identifier *N* that is also preloaded from the local *env* table.

```
j1 joinEvent(@NI,E) :- periodic(@NI,E,1,2).
j2 join(@NI,E) :- joinEvent(@NI,E).
j3 joinReq@LI(LI,N,NI,E) :- joinEvent(@NI,E), nodeID(@NI,N),
                                  landmark(@NI,LI), LI != “NIL”.
j4 succ(@NI,N,NI) :- landmark(@NI,LI), joinEvent(@NI,E),
                                  nodeID(@NI,N), LI = “NIL”.
j5 lookup@LI(LI,N,NI,E) :- joinReq@LI(LI,N,NI,E).
j6 succ(@NI,S,SI) :- join(@NI,E), lookupResults(@NI,K,S,SI,E).
```

Figure 6.9: Rules for joining the Chord ring.

Figure 6.9 shows the rules for joining the Chord ring. To enter the ring, a node *NI* generates a *joinEvent* tuple locally (rule j1) whose arrival triggers rules j2-j6. Rule j2 creates a *join* tuple upon the arrival of the *joinEvent* tuple. In rule j3, if the landmark node is known (i.e., not NIL), a *joinReq* tuple is sent to the landmark node; otherwise rule j4 sets the node to point to itself as a successor, forming an overlay by itself and awaiting others to join in. When the landmark receives a *joinReq* tuple, rule j5 initiates a lookup from the landmark node for the successor of the joining node's identifier *N*, and set the return address of the lookup to be *NI*. If the lookup is successful, a *lookupResults* event is received at node *NI*. Rule j6 then defines the joining node's successor (*succ* table) to be the result of the lookup.

6.3.3 Chord Ring Maintenance

```
sb1 succ(@NI,P,PI) :- periodic(@NI,E,10), nodeID(@NI,N),
                      bestSucc(@NI,S,SI), pred(@SI,P,PI),
                      PI != “NIL”, P in (N,S).
sb2 succ(@NI,S1,SI1) :- periodic(@NI,E,10), succ(@NI,S,SI), succ(@SI,S1,SI1).
sb3 pred@SI(SI,N,NI) :- periodic(@NI,E,10), nodeID(@NI,N),
                      succ(@NI,S,SI), pred(@SI,P,PI),
                      node(@SI,N1), ((PI = “NIL”)  (N in (P,N1))).
```

Figure 6.10: Rules for ring stabilization.

After joining the Chord network, each node performs the ring maintenance protocol in order to maintain a set of successors and a single predecessor. Candidate successors (and the single predecessor) are found during the *stabilization* phase of the Chord overlay maintenance. The rules specifying the stabilization phase in Figure 6.10. Stabilization is done periodically at time intervals of 15 seconds by the rules sb1, *sb2* and *sb3*. Rule sb1 ensures that a node's best successor's predecessor is also stored in its successor table. In rule sb2, each successor periodically asks all of its successors to send it their own successors. In rule sb3, a node periodically notifies its successors about itself, allowing its successors to point their respective predecessors to the notifying node if it is closer in key-space compared to their current predecessors.

```
n1 newSuccEvent(@NI) :- succ(@NI,S,SI).
n2 newSuccEvent(@NI) :- deleteSucc(@NI,S,SI).
n3 bestSuccDist(@NI,min<D>) :- newSuccEvent(@NI),nodeID(@NI,N),
                               succ(@NI,S,SI), D = S - N - 1.
n4 bestSucc(@NI,S,SI) :- succ(@NI,S,SI), bestSuccDist(@NI,D), nodeID(@NI,N),
                               D = S - N - 1.
n5 finger(@NI,0,S,SI) :- bestSucc(@NI,S,SI).
```

Figure 6.11: Rules for computing best successor and first finger entry.

Based on the set of candidate successors obtained from stabilization, additional rules are required in order to select the best successor, and also evict successors that are no longer required. In Figure 6.11, rule n1 generates a *newSuccEvent* event tuple upon the insertion (refresh) of a new (existing) successor. Rule n2 generates a *newSuccEvent* for deletions of an existing successor. The *newSuccEvent* event tuple triggers rules n3 and *n4*, which are used to define as "best" the successor among those stored in the *succ* stored table whose identifier distance from the current node's identifier is the lowest. Rule n5 further ensures that the first finger entry (used for routing lookups) is always the same as the best successor.

As new successors are discovered, successor selection only keeps those successors closest to a node in the table, evicting at each discovery the single remaining node (rules s1-s4 in Figure 6.12).

```
s1 succCount(@NI,count<*>) :- newSuccEvent(@NI), succ(@NI,S,SI).
s2 evictSucc(@NI) :- succCount(@NI,C), C > 4.
s3 maxSuccDist(@NI,max<D>) :- succ(@NI,S,SI),
                          nodeID(@NI,N), evictSucc(@NI),
                          D = S - N - 1.
s4 delete succ(@NI,S,SI) :- nodeID(@NI,N), succ(@NI,S,SI),
                          maxSuccDist(@NI,D), D = S - N - 1.
```

Figure 6.12: Rules for successor selection.

6.3.4 Finger Maintenance and Routing

```
l1 lookupResults(@R,K,S,SI,E) :- nodeID(@NI,N), lookup(@NI,K,R,E),
                          bestSucc(@NI,S,SI), K in (N,S].
l2 bestLookupDist(@NI,K,R,E,min<D>) :- nodeID(@NI,N),
                          lookup(@NI,K,R,E), finger(@NI,I,B,BI),
                          D = K - B - 1, B in (N,K).
l3 lookup(min<@BI>,K,R,E) :- nodeID(@NI,N),
                          bestLookupDist(@NI,K,R,E,D), finger(@NI,I,B,BI),
                          D = K - B - 1, B in (N,K).
```

Figure 6.13: Rules for recursive lookups in Chord

The *finger* table is used in Chord to route lookup requests. Figure 6.13 shows the three rules that are used to implement lookups in Chord. Each *lookup(@NI,K,R,E)* event tuple denotes a lookup request at node *NI* for key *K*, originates from node *R* with event identifier *E*.

From our earlier introduction to the Chord protocol, we note that all lookup requests for key *K* seek the node whose identifier is the immediate successor on the ring of *K*. Rule l1 is the base case, returning a successful lookup result if the received lookup seeks a key *K* found between the receiving node's identifier and that of its best successor (we come back to the best successor below). Rule *l2* is used in non-base cases, to find the minimum distance (in key identifier space modulo 2^{160}) from the local node's fingers to *K* for every finger node *BI* whose identifier *B* lies between the local node's identifier *N* and *K*. Rule l3 then selects one of the finger entries with the minimum distance to key *K* as the target node *BI* to receive the lookup request. Since there can be multiple such finger entries, the *min¡BI¿* aggregate ensures that only one of the finger entries receives the forwarded lookup.

Figure 6.14 shows the rules for generating the entries in the *finger* table. There are two additional materialized tables *fFix* and *nextFingerFix* that store intermediate state for the finger fixing protocol. The *nextFingerFix* table stores one tuple *nextFingerFix(@NI,I)* that stores the next finger entry *I* to be picked for fixing at node *NI*.

Every 10 seconds, rule f1 selects the *I* finger to fix, and then generates a *fFix(@NI,E,I)* tuple that denotes that the *I* finger is selected for fixing with event identifier *E*. This results in the generating of a *fFixEvent(@NI,E,I)* event tuple in rule *f2* which will generate a *lookup* request for key $K = 2^I + N$ with the corresponding event identifier *E*. When the lookup succeeds, rule f4 receives a *lookupResults* event tuple, which it then uses to update all the corresponding finger entries (*f5-6*). Rules f7-f9 then deletes the *fFix* tuple, and then increments the *I* field of *nextFingerFix* by 1 for fixing the next finger entry in the next period. Rule f10 sets the *uniqueFinger* based on new *finger* entries.

6.3.5 Failure Detection

Figure 6.15 shows the rules that a node utilizes for sending keep-alive messages to its neighbors. The rules are similar to that of the *Ping-Pong* program presented in Chapter 2.

```
f1 fFix(@NI,E,I) :- periodic(@NI,E,10), nextFingerFix(@NI,I).
f2 fFixEvent(@NI,E,I) :- fFix(@NI,E,I).
f3 lookup(@NI,K,NI,E) :- fFixEvent(@NI,E,I), nodeID(@NI,N), K = 0x1I << I + N.
f4 eagerFinger(@NI,I,B,BI) :- fFix(@NI,E,I), lookupResults(@NI,K,B,BI,E).
f5 finger(@NI,I,B,BI) :- eagerFinger(@NI,I,B,BI).
f6 eagerFinger(@NI,I,B,BI) :- nodeID(@NI,N),
                              eagerFinger(@NI,I1,B,BI), I = I1 + 1,
                              K = 0x1I << I + N, K in (N,B), BI != NI.
f7 delete fFix(@NI,E,I1) :- eagerFinger(@NI,I,B,BI), fFix(@NI,E,I1),
                              I > 0, I1 = I - 1.
f8 nextFingerFix(@NI,0) :- eagerFinger(@NI,I,B,BI), ((I = 159)  (BI = NI)).
f9 nextFingerFix(@NI,I) :- nodeID(@NI,N),
                              eagerFinger(@NI,I1,B,BI), I = I1 + 1,
                              K = 0x1I << I + N, K in (B,N), NI != BI.
f10 uniqueFinger(@NI,BI) :- finger(@NI,I,B,BI).
```

Figure 6.14: Rules for generating finger entries.

At regular intervals of 5 seconds, each node generates one *pendingPing* tuple for each one of its neighbors (rules pp1, *pp2* and *pp3*). This results in *pingReq* messages that are periodically (every 3 seconds as indicated in rule pp3) sent to the respective neighbors for the lifetime of each *pendingPing* tuple. These *pendingPings* are deleted upon receiving the corresponding *pingResp* messages.

Figure 6.16 shows the rules for detecting failure of successors, predecessors and fingers. Here, rule fd1 generates *nodeFailure* events when there are outstanding *pendingPing* tuples that are unanswered after a period of time. The choices of 7 seconds in rule fd1 and 3 seconds in rule pp5 determine the frequency in which *pingReq* messages are sent, and the

```
pp1 pendingPing(@NI,SI,E1,T) :- periodic(@NI,E,5), succ(@NI,S,SI),
                                E1 = f_rand(), SI != NI, T = f_now().
pp2 pendingPing(@NI,PI,E1,T) :- periodic(@NI,E,5), pred(@NI,P,PI),
                                E1 = f_rand(), PI ! = “NIL”, T = f_now().
pp3 pendingPing(@NI,FI,E1,T) :- periodic(@NI,E,5), uniqueFinger(@NI,FI),
                                E1:=f_rand(), T:=f_now().
pp4 pingResp(@RI,NI,E) :- pingReq(@NI,RI,E).
pp5 pingReq(@PI,NI,E) :- periodic(@NI,E1,3),
                                pendingPing(@NI,PI,E,T).
pp6 delete pendingPing(@NI,SI,E,T) :- pingResp(@NI,SI,E), pendingPing(@NI,SI,E,T).
```

Figure 6.15: Rules for sending keep-alives.

number unanswered replies that are required before we conclude a node is “dead”. In our example, a node is considered “dead” if there are two successive unanswered *pingReq* messages. The *nodeFailure* event then results in deletion of *pendingPing*, *succ* and *finger* entries, and resetting the single *pred* entry (rules fd3-fd7). A *deleteSucc* event is generated to allow the recomputation of the best successor in rules n2-n5.

6.3.6 Summary of Chord

In this section, we have presented the *NDlog* rules that are necessary for implementing the Chord distributed hash table. In doing so, we have also demonstrated the compactness of *NDlog*: Chord is specified in only 48 rules, which is two orders of magnitude less code compared to an equivalent C++ implementation [85]. In addition, in order to deal with issues related to message delivery, acknowledgments, failure detection and timeouts, we have made extensive use of soft-state tables and soft-state rules that were presented in

```
fd1 nodeFailure(@NI,PI,E1,D) :- periodic(@NI,E,1), pendingPing(@NI,PI,E1,T),
                                T1 = f_now(), D = T-T1, D > 7.
fd2 delete pendingPing(@NI,PI,E,T) :- nodeFailure(@NI,PI,E,D),
                                pendingPing(@NI,PI,E,T).
fd3 deleteSucc(@NI,S,SI) :- succ(@NI,S,SI), nodeFailure(@NI,SI,E,D).
fd4 delete succ(@NI,S,SI) :- deleteSucc(@NI,S,SI).
fd5 pred(@NI,"NIL","NIL") :- pred(@NI,P,PI), nodeFailure(@NI,PI,E,D).
fd6 delete finger(@NI,I,B,BI) :- finger(@NI,I,B,BI), nodeFailure(@NI,BI,E,D).
fd7 delete uniqueFinger(@NI,FI) :- uniqueFinger(@NI,FI), nodeFailure(@NI,FI,E,D).
```

Figure 6.16: Rules for failure detection of successors, predecessors and fingers.

Chapter 2

As a summary, in addition to the *materialize* statements and initialization rules i1-i4, we review our *NDlog* rules based on the three functionalities of a typical overlay network that we presented earlier:

- **Overlay formation and maintenance:** rules j1-j6 are used by a node joining the Chord network via a landmark. Once a node has joined the ring, rules sb1-sb3 are used to execute the ring stabilization to learn about new successors and refine the predecessor. Based on the successors learned, rules n1-n5 are used for selecting the best successor, and rules s1-s4 are used for evicting unnecessary successors. To ensure that all overlay neighbors are alive, rules pp1-pp5 and *fd1-fd7* for periodically pinging all successors, predecessors and finger entries, and deleting them if they do not respond to heartbeats.

- **Routing:** Given the basic ring network, rules f1-f9 for generating finger table entries that ensures scalable lookups.

- **Forwarding:** With the finger table in place, rules l1-l3 are used for routing lookup requests via the finger table.

We note that *overlay formation* constitutes the majority of P2-Chord rules, and clearly illustrates the additional challenges in specifying declarative overlays compared to the relatively simpler *NDlog* programs for implementing routing protocols presented in Chapter 3.

6.4 Summary

In this chapter, we demonstrated the use of *NDlog* for expressing two complex overlay networks, namely the Narada mesh formation and a full-fledged implementation of the Chord distributed hash table in 16 and 48 rules respectively.

We note that our Chord implementation is roughly two orders of magnitude less code than the original C++ implementation. This is a quantitative difference that is sufficiently large that it becomes qualitative: in our opinion (and experience), declarative programs that are a few dozen lines of code are markedly easier to understand, debug and extend than multi-thousand-line imperative programs. In the next chapter, we present experimental results that validate the correctness of our *NDlog* programs for Narada and Chord.

Chapter 7

Evaluation

In this chapter, we evaluate the performance of P2's implementation of *NDlog*. Our experiments take as input *NDlog* programs, and compile them into P2 dataflows as described in Chapter 5. These dataflows are then executed using the P2 system. As our experimental testbed, we make use of Emulab [40], a cluster-based testbed that supports realistic emulation of latency and bandwidth constraints seen on the Internet, while providing repeatable experiments in a controlled environment.

The goal of our evaluation is twofold. First, we aim to validate that our declarative specifications result in the expected network properties in terms of topology and messaging. Second, we examine our raw performance with an eye toward feasibility: we do not expect our per-node performance to be as good as a highly-tuned hand-coded implementation, but we would like it to be acceptable and exhibit scalability trends that one would expect.

The chapter is organized as follows. In Section 7.1 we present our evaluation of the path vector protocol. In Section 7.2, we present evaluations of the *Narada mesh* and the *Chord distributed hash table*.

7.1 Declarative Routing Evaluation

In our first experiment, we evaluate the performance of declarative routing protocols written in *NDlog* using the P2 system. The main metrics that we use in our evaluation are:

Convergence time: Given a quiesced network, the time taken for the network protocol to generate all its eventual network state. This is equivalent to achieving *fixpoint* during *NDlog* program execution, where there are no new derivations from all rules that are being executed.

Communication overhead: The number of bytes transferred for each network protocol in order to achieve convergence in a quiesced network. We consider both aggregate communication overhead (MB), as well as per-node bandwidth (KBps).

As input to the Emulab testbed, we use transit-stub topologies generated using GT-ITM [53], a package that is widely used to model Internet topologies. Our topology has four transit nodes, eight nodes per stub and three stubs per transit node. Latency between transit nodes is 50 ms, latency between transit nodes and their stub nodes is 10 ms, and latency between any two nodes in the same stub is 2 ms. The link capacity is set to 10 Mbps. Given the small size of our network, we limited our topology to four transit domains.

We construct an overlay network over the base GT-ITM topology where each overlay node is assigned to one of the stub nodes. Each overlay node runs the P2 system on one Emulab machine, and picks four randomly selected overlay neighbors which are stored as facts in each local *#link* table.

7.1.1 Scalability of Path-Vector Protocol

In our first experiment, we measure the performance of our system when all nodes are running the *Shortest-Path* program of Chapter 2, which implements the path-vector protocol used to compute the shortest latency paths between all pairs of nodes. In our implementa-

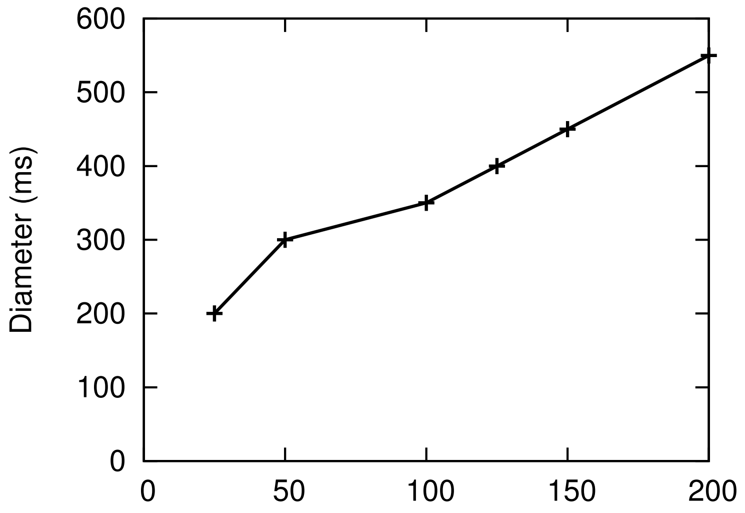

Figure 7.1: Network diameter (ms) vs Number of nodes.

tion, we use the *aggregate selections* optimization to avoid sending redundant path tuples (Section 8.1.1), where the most recently computed shortest paths are batched and sent to neighboring nodes every 500 ms. The duration of 500 ms is chosen as it is an upper bound on the latency between any two nodes. This ensures that computed paths at each iteration have sufficient time to be propagated and accumulated at every node for periodic aggregate selections to be most effective.

Figure 7.1 shows the diameter of the network (computed from the maximum of all shortest-path latencies) used in our experiments as the number of nodes increases from 25 to 200. Figures 7.2 and 7.3 show the convergence latency and per-node communication overhead for the *Shortest-Path* program as the number of nodes increases. We make two observations.

- The convergence latency for the *Shortest-Path* program is proportional to the network diameter. This is expected because in a static network, the convergence time of the path vector protocol depends on the time taken to compute the *longest* shortest paths,

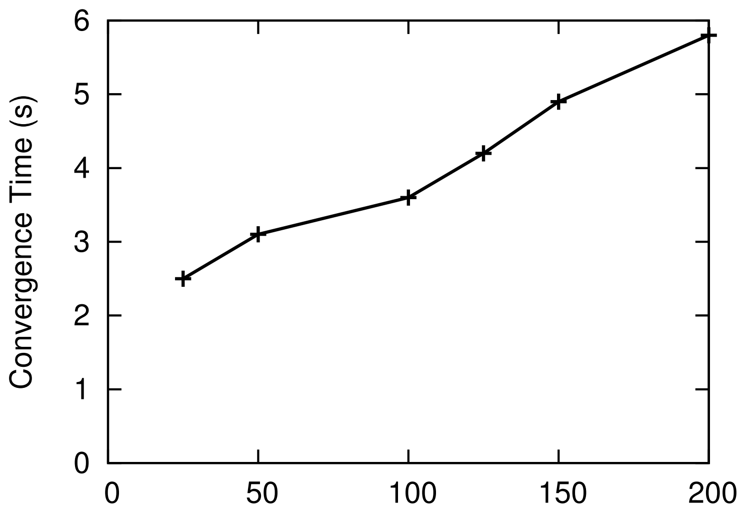

Figure 7.2: Convergence latency (s) vs Number of nodes.

which in our case is bounded by the time taken for the computed shortest paths to propagate in the network (*i.e.*, $500ms \times D_{hop}$, where D_{hop} is the network diameter in terms of hop count).

- The per-node communication overhead increases linearly with the number of nodes. This is because each node needs to compute the shortest path to every other node in the network.

We note that both these observations are consistent with the scalability properties of the traditional distance vector and path vector protocols, suggesting that our approach does not introduce any fundamental overheads when used to implement traditional routing protocols.

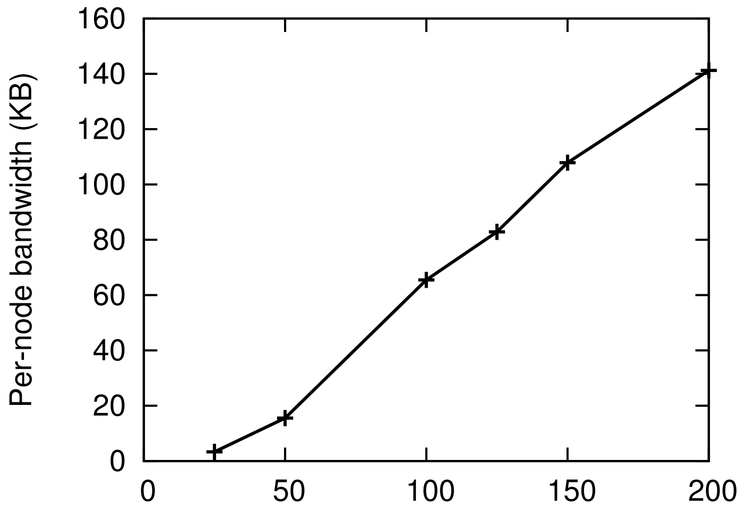

Figure 7.3: Per-node Communication Overhead (KB).

7.1.2 Incremental Evaluation in Dynamic Networks

In our next experiment, we examine the overhead of incrementally maintaining *NDlog* program results in a dynamic network. We run the same *Shortest-Path* program on 100 Emulab nodes over a period of time, and subject the network to bursty updates as described in Section 5.4. Each update burst involves randomly selecting 10% of all links, and then updating the cost metric by up to 10%.

We use the shortest-path random metric since executing the *NDlog* program using this metric is most demanding in terms of bandwidth usage and convergence time. This is because as we noted in Section 8.1.1, aggregate selections are most useful for queries whose input tuples tend to arrive over the network out of order in terms of the monotonic aggregate – *e.g.*, computing "shortest" paths for metrics that are not correlated with the network delays that dictate the arrival of the tuples during execution.

Figure 7.4 plots the per-node communication overhead, when applying a batch of up-

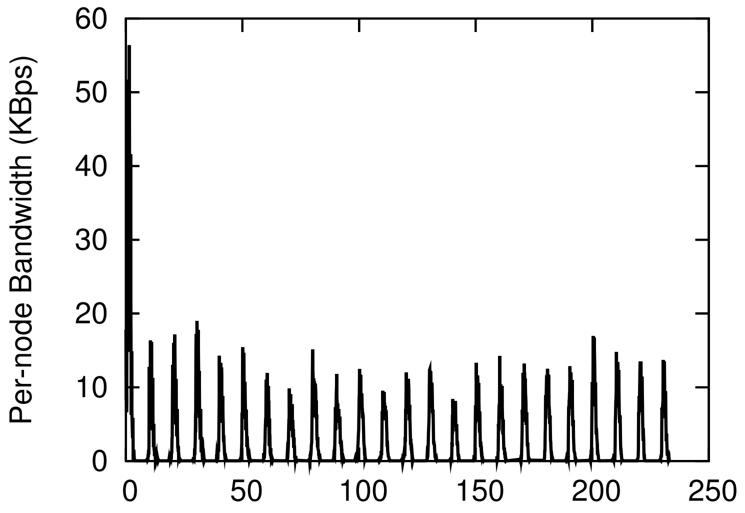

Figure 7.4: Per-node Bandwidth (KBps) for periodic link updates on latency metric (10s update interval).

dates every 10 seconds. Two points are worth noting. First, the time it takes the program to converge after a burst of updates is well within the convergence time of running the program from scratch. This is reflected in the communication overhead, which increases sharply after a burst of updates is applied, but then disappears long before the next burst of updates (Figure 7.4). Second, each burst peaks at 19 KBps, which is only 32% of the peak bandwidth and 28% of the aggregate bandwidth of the original computation. Our results clearly demonstrate the usefulness of performing incremental evaluation in response to changes in the network, as opposed to recomputing the queries from scratch

We repeat our experiment using a more demanding update workload (Figure 7.5), where we interleave update intervals that are 2 seconds and 8 seconds, the former interval being less than the from-scratch convergence time of 3.6 seconds. We observe that despite the fact that bursts are sometimes occurring faster than queries can run, bandwidth usage is similar to the less demanding update workload, peaking at 24 KBps and converging within

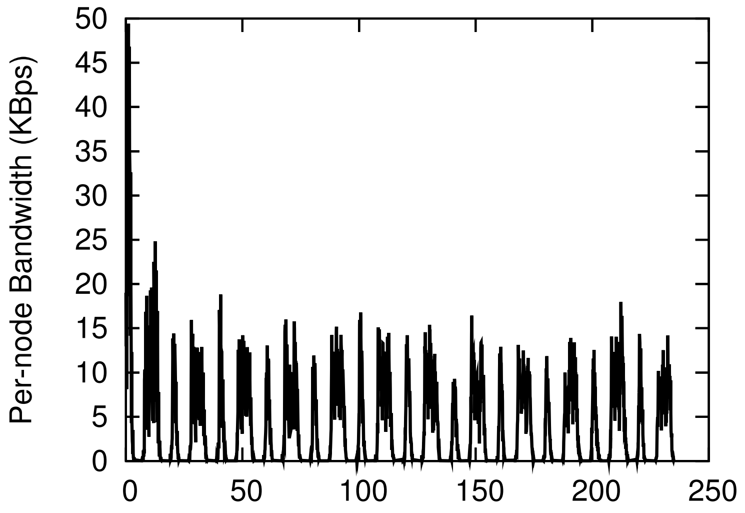

Figure 7.5: Per-node Bandwidth (KBps) for periodic link updates (interleaving 2s and 8s update interval).

the from-scratch convergence time.

7.2 Declarative Overlays Evaluation

In this section, we present performance results of the Narada mesh and the Chord DHT. Our experiments are carried out on 100 machines on the Emulab testbed [40]. In both overlay networks, the latency between any two overlay nodes is set to 100 ms, and link capacity is set to 10 MBps.

7.2.1 Narada Mesh Formation

We evaluate the Narada specifications on mesh formation shown in Appendix B.1. Our experiment consists of 100 Narada nodes, one on each Emulab node. All nodes join the network over a span of 10 seconds. Each Narada node has an initial set of neighbors, and at regular intervals of 5 seconds, propagate its entire membership list to its neighbors. We measure the per-node bandwidth (KBps) of periodically sending the membership list in the steady state, and also the convergence time (seconds) taken for all Narada nodes have achieved full membership knowledge of the entire network.

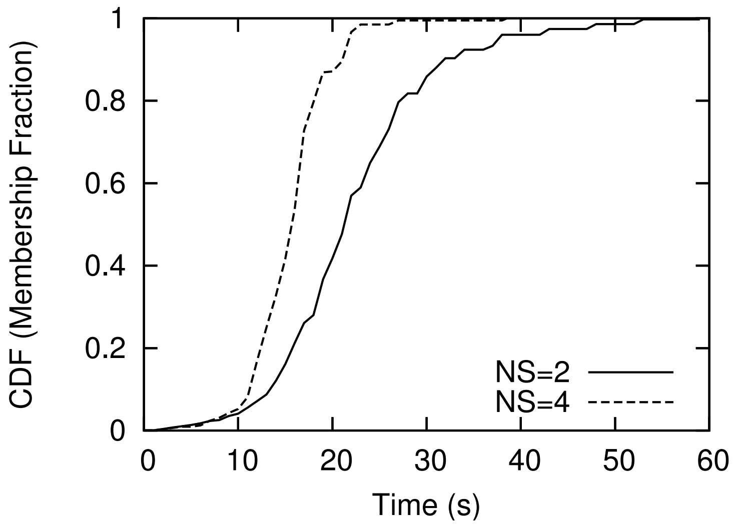

Figure 7.6: CDF of average Narada membership at each node as fraction of total network size over time (s).

Figure 7.6 shows the CDF of membership at each node as a fraction of the entire network size over time (seconds) for a network size of 100 for two experimental runs (*NS=2, NS=4*) where we vary the number of neighbors that each node has (2 and 4 neighbors). Each data point (x,y) shows the average fraction y of the network that each node knows at time x. Upon convergence, all nodes learn about every other node in the network (*i.e.*, $y = 1$).

Our results show that our Narada implementation converges on the sparser network ($NS = 2$) within 60 seconds, while requiring less than 40 seconds to converge on the denser network ($NS = 4$). The convergence time includes the initial 10 seconds as nodes join the Narada network. Our results clearly demonstrate the tradeoffs between bandwidth and convergence in propagating the membership list. The faster convergence of the denser network comes at the expense of bandwidth utilization ($33KBps$) as compared to $13KBps$ for the sparser network.

7.2.2 P2-Chord

In this section, we focus on measuring the full Chord DHT specification in Appendix B.2. Chord is a good stress test of our architecture, being relatively complex compared to other overlay examples like gossip and end-system multicast. Chord also has the advantage of being well-studied. Our P2-Chord deployment on the Emulab testbed [40] consists of 100 machines executing up to 500 P2-Chord instances (5 P2 processes running on each Emulab machine). We utilize the same network topology as the Narada experiment. Since we are running up to 5 P2 processes per Emulab node, we selected Emulab machines with newer hardware (64-bit Xeon 3000 series with 2 GB memory) to run our experiments.

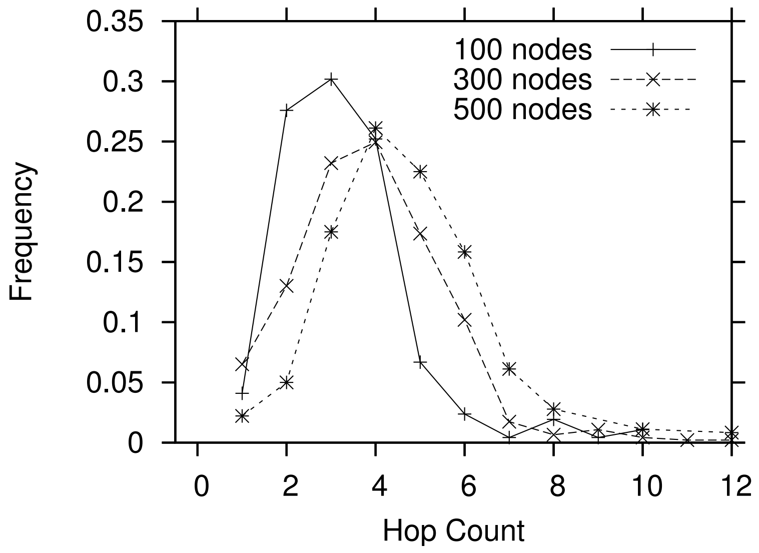

Figure 7.7: Hop-count distribution for lookups.

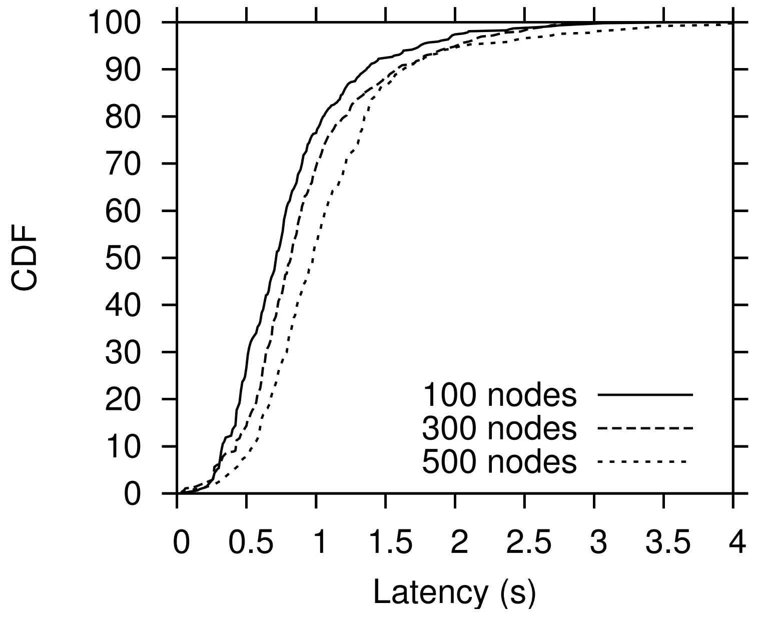

Figure 7.8: CDF for lookup latency.

7.2.2.1 Static Network Validation

In our first round of experiment, we validate the high-level characteristics of the Chord overlay. We generate a uniform workload of DHT "lookup" requests to a static set of nodes in the overlay, with no nodes joining or leaving. This is somewhat unrealistic but it allows us to ensure we are achieving the static properties of Chord. In each experiment, we start a landmark node, and have all other nodes join the landmark node at regular intervals. Once all the nodes have joined the Chord overlay, we issue lookups every 15 seconds simultaneously (with the same lookup key K) from 10 nodes.

Figure 7.7 shows the hop-count distribution for our workload. Except for a few outliers, 99% of all lookups complete within 10 hops. The average hop count of lookups are 3.3, 4.0 and 4.5 for node sizes of 100, 300 and 500 respectively, approximating the theoretical average of $0.5 \times log_2(N)$, where N is the number of nodes.

Figure 7.8 shows the CDF of lookup latencies for different network sizes. As expected, the average latency increases in proportion to the average lookup hop count for each network size. On a 500 node static network, 99% of all lookups complete in less than 3.4 seconds. The average (median) latencies are 0.81 seconds (0.72 seconds), 0.92 seconds (0.82 seconds) and 1.09 seconds (0.98 seconds) for node sizes of 100, 300 and 500 respectively. Our average and median latency numbers are within the same order of magnitude as the published numbers [114] of the MIT Chord deployment.

In addition to achieving expected latency numbers, our lookups are also "correct". All lookup requests return successfully with the lookup requests. In addition, all lookups achieve 100% consistency, where all lookup requests for the same key issued from different nodes return identical results.

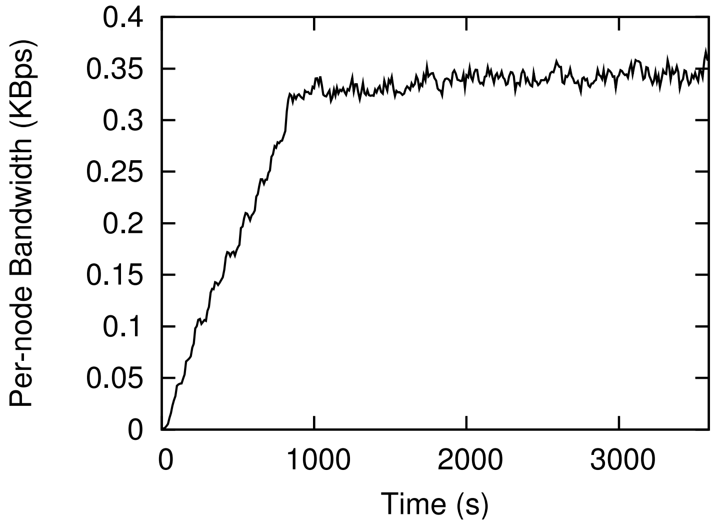

Figure 7.9: Per-node Bandwidth (KBps) over time (s).

Figure 7.9 shows the per-node bandwidth (KBps) consumption over time (in seconds) for a static P2-Chord network where fingers are fixed every 10 seconds, and ring stabilization (exchange of successors and predecessors among neighbors) happen every 10 seconds.

Each node periodically send ping messages to neighbors every 3 seconds. After an initial linear increase in bandwidth as nodes join the Chord ring, the bandwidth utilization stabilizes at 0.34 KBps, well within the published bandwidth consumption of 1 KBps [103] of other high consistency and low latency DHTs.

7.2.2.2 Churn Performance

In our second round of experiments, we focus on the performance of our Chord implementation under varying degrees of membership churn. Again, our goal is to validate that our compact specification of Chord faithfully captures its salient properties following the methodology in [103]. We bring up a 100 node Chord network, and once the network is stable, induce churn on the network for 20 minutes as follows. Periodically, we select a node at random to fail. Upon each node failure, a new node immediately joins the Chord network with a different node identifier. We vary the interval between every node failure/restart event to achieve different average node session times (8, 16, 47 and 90 minutes).

In the steady state under constant churn, we issued lookups for the same key simultaneously from 10 different nodes every 15 seconds. Following the methodology in [103], we define a *consistent* lookup when a majority of the lookups (>5) see a consistent result that points to the same node that owns the key. For each group of 10 lookups, we compute the maximum fraction of lookups that share a consistent result. P2-Chord's churn parameters are set as follows: (1) the fix finger and ring-stabilization periods are both set to 10 seconds as before; (2) Each node periodically send ping messages to neighbor nodes every 3 seconds, and remove entries from the local neighbor tables if they do not respond to two successive pings.

Figure 7.10 shows the CDF (log-scale for Y-axis) for the *consistent fraction* of lookups, which is defined as the fraction of lookups with consistent result for each group of simultaneous lookups. To interpret the graph, each data-point (x, y) shows the fraction y of lookups with lookup consistency less than x. Our results show that P2 Chord does well under low churn (session times of 90 minutes and 47 minutes), generating 99% and 96% consistent lookups. Under high churn (session times of 16 minutes and 8 minutes), P2 Chord performs well, producing 95% and 79% consistent lookups.

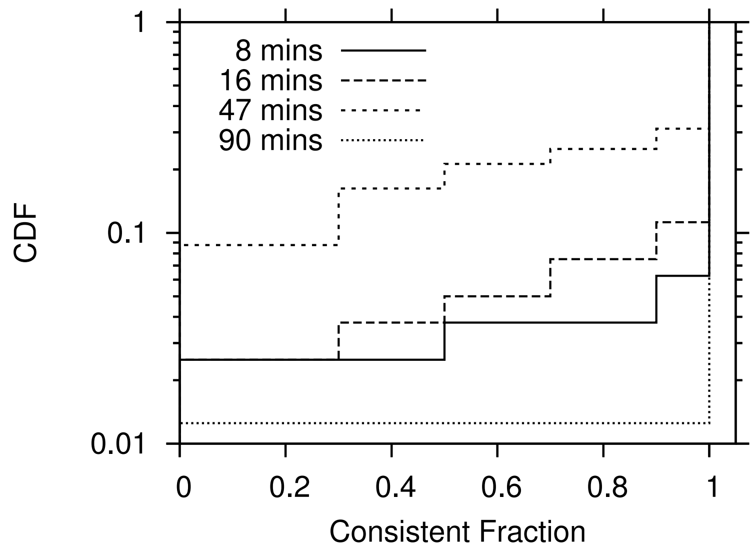

Figure 7.10: CDF for lookup consistency fraction under churn.

Figure 7.11 shows the CDF of the lookup latencies for diferent churn rates. At low churn rates, the lookup latencies are similar to those measured under a stable Chord network with no churn. At high churn rates, the average lookup latency increased from 0.81 seconds to 1.01 seconds and 1.32 seconds respectively.

While P2-Chord performs acceptably, it clearly does not attain the published figures for the MIT implementation (at least 99.9% consistency for a session time of 47 minutes). Ultimately, an evaluation of a system like P2 rests on an assessment of the ideal tradeoff between code size and performance. It may be the case that churn performance can be at the expense of additional rules that implements lookup retries on a per-hop basis, and better failure detection techniques with adaptive timers.

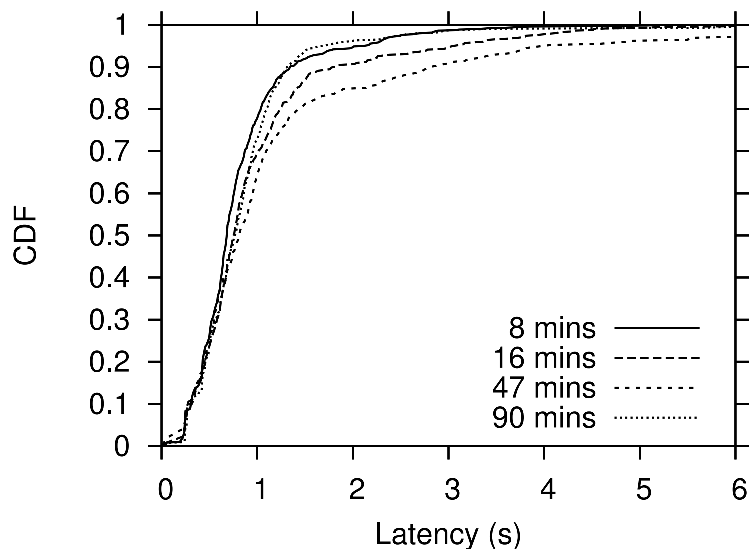

Figure 7.11: CDF for lookup latency under churn.

7.3 Summary

In this chapter, we evaluated a variety of declarative networking protocols on the Emulab testbed. We demonstrated that (1) *NDlog* programs are faithful to their specifications, and (2) the P2 system correctly executed the declarative specifications to achieve the correct network implementation. We based our correctness criteria of a declarative network on its ability to achieve expected network properties in terms of topology and messaging, and also to incur maintenance overhead and forwarding performance that were within published results of other equivalent implementations.

In summary, we made the following observations from our evaluation results:

- Our *Shortest-Path NDlog* program exhibited scalability trends similar to that of traditional distance vector and path vector protocols. We observed that the convergence time of this program was proportional to the network diameter, and generated communication overhead that was linear to the number of nodes. This validated our routing protocol and demonstrated that there were no fundamental overheads in our approach relative to traditional approaches.

- We further demonstrated the usefulness of performing incremental evaluation of the *Shortest-Path NDlog* program in a dynamic network, where the shortest paths was computed at a bandwidth cost that was a fraction of recomputing the programs from scratch. As in network protocols, such incremental evaluation is required both for timely updates and for avoiding the overhead of recomputing all routing tables whenever there are changes to the underlying network.

- The *NDlog* declarative overlay programs for the Narada mesh and Chord achieve the expected high-level properties of their respective overlay networks for both static and dynamic networks. For example, in a static network of up to 500 nodes, the measured hop-count of lookup requests in the Chord network conformed to the theoretical average of $0.5 \times log_2N$ hops, and the latency numbers were within the same order of magnitude as published Chord numbers. The steady state maintenance overhead was also within the published bandwidth consumption of other high consistency and low latency DHTs. In a dynamic network, our Chord implementation was able to achieve good lookup performance under low churn and respectable performance under high churn.

In the next chapter, we describe a number of optimizations that are useful in the declarative networking setting, and present evaluation results to validate the effectiveness of these optimizations.

Chapter 8

Optimization of NDlog Programs

One of the promises of a declarative approach to networking is that it can enable automatic optimizations of protocols, much as relational databases can automatically optimize queries. These not only reduces the burden on programmers, it also enables what Codd called *data independence* [33]: the ability for the implementation of a program to adapt to different underlying execution substrates.

Our main goals in this chapter are to demonstrate that our declarative approach is amenable to automatic query optimizations, and to illustrate the close connection between network optimizations and query optimizations. In doing so, we open up what appears to be a rich new set of research opportunities.

The chapter is organized as follows. In Section 8.1, we explore the application of traditional Datalog optimizations in the declarative networking context. We then propose new techniques for multi-query optimizations and cost-based optimizations in Sections 8.2 and 8.3 respectively. To validate our proposed optimizations, in Section 8.4, we present our measurements of the performance of the P2 system executing optimized declarative routing queries on the Emulab testbed.

8.1 Traditional Datalog Optimizations

We first explore the applicability of three traditional Datalog optimization techniques: *aggregate selections*, *magic sets* and *predicate reordering*. We focus primarily on optimizing declarative routing queries which are generally variants of transitive closure queries. There have been substantial previous work on optimizing such queries in centralized settings. In Section 10.3, we discuss how these optimization techniques can be extended to support more complex declarative overlay networks.

8.1.1 Aggregate Selections

A naïve execution of the *Shortest-path* program computes all possible paths, even those paths that do not contribute to the eventual shortest paths. This inefficiency can be avoided with an optimization technique known as *aggregate selections* [118; 47].

Aggregate selections are useful when the running state of a monotonic aggregate function can be used to prune program evaluation. For example, by applying aggregate selections to the *Shortest-path* program, each node only needs to propagate the current shortest paths for each destination to neighbors. This propagation can be done whenever a shorter path is derived.

A potential problem with this approach is that the propagation of new shortest paths may be unnecessarily aggressive, resulting in wasted communication. As an enhancement, we propose a modified scheme, called *periodic aggregate selections*, where a node buffers up new paths received from neighbors, recomputes any new shortest paths incrementally, and then propagates the new shortest paths periodically. The periodic technique has the potential for reducing network bandwidth consumption, at the expense of increasing convergence time. It is useful for queries whose input tuples tend to arrive over the network in an order that is not positively correlated with the monotonic aggregate – *e.g.*, computing "shortest" paths for metrics that are not correlated with the network delays that dictate the arrival of the tuples during execution.

In addition, aggregate selections are necessary for the termination of some queries. For example, with aggregate selections, even if paths with cycles are permitted, the *Shortest-Path* program will terminate, avoiding cyclic paths of increasing lengths.

8.1.2 Magic Sets and Predicate Reordering

The *Shortest-Path* program in our example computes *all-pairs* shortest paths. This leads to unnecessary overhead when querying for paths between a limited set of sources and/or destinations. This problem can be alleviated by applying two optimization techniques: *magic-sets rewriting* and *predicate reordering*.

Magic-Sets Rewriting: To limit computation to the relevant portion of the network, we use a query rewrite technique called *magic sets rewriting* [16; 18]. The Magic Sets method is closely related to methods such as Alexander [106] and QSQ [71], all of which are designed to avoid computing facts that do not contribute to the final answer to a recursive query. The proposed processing techniques in Chapter 5 are based on bottom-up (or forward-chaining) evaluation [96] where the bodies of the rules are evaluated to derive the heads. This has the advantage of permitting set-oriented optimizations while avoiding infinite recursive loops, but may result in computing redundant facts not required by the program. For example, even when the *Shortest-Path* program (Figure 2.4 in Chapter 2) specifies *shortestPath(@a,b,Z,P,C)* as the "goal" of the query, naïvely applying bottom-up evaluation results in the computation of *all* paths between *all* pairs of nodes.

The magic sets rewrite avoids these redundant computations and yet retains the two advantages of bottom-up evaluation. The key ideas behind the rewrite include (1) the introduction of "magic predicates" to represent variable bindings in queries that a top-down search would ask, and (2) the use of "supplementary predicates" to represent how answers are passed from left-to-right in a rule. The rewritten program is still evaluated in a bottom-up fashion, but the additional predicates generated during the rewrite ensure that there are no redundant computations.

We illustrate the use of magic sets in an example: by modifying rule sp1 from the *Shortest-Path* program, the following program in Figure 8.1 computes only those paths leading to destinations in the *magicDst* table.

```
#include(sp2,sp3,sp4)
sp1-d path(@S,D,D,P,C) :- magicDst(@D),#link(@S,D,C), P = f_init(S,D).
m1 magicDst(@a).
Query shortestPath(@S,a,P,C).
```

Figure 8.1: Shortest-Path program with magic sets

Rule sp1-d initializes 1-hop paths for destinations whose *magicDst(@D)* is present in the *magicDst* table. Rule m1 adds a *magicDst(@a)* fact in the *magicDst* table. Intuitively, the set of *magicDst(@D)* facts is used as a "magic predicate" or "filter" in the rules defining paths. This ensures that rule sp2 propagates paths to selected destinations based on the *magicDst* table (in this case, paths to only node *a*). The shortest paths are then computed as before using rules sp3 and sp4.

Predicate Reordering: The use of magic sets in the previous program is not useful for pruning paths from sources. This is because paths are derived in a *"Bottom-Up" (BU)* fashion starting from destination nodes, where the derived paths are shipped "backwards" along neighbor links from destinations to sources. Interestingly, switching the search strategy can be done simply by *reordering* the *path* and #link predicates. Recall from Chapter 2 that predicates in a rule are evaluated in a default left-to-right order. This has the effect of turning sp2 from a *right-recursive* to a *left-recursive* rule: the recursive predicate is now to the left of the non-recursive predicate in the rule body. Together with the use of magic sets, the *Magic-Shortest-Path* program in Figure 8.2 allows filtering on *both* sources and *destinations*, as we proceed to describe.

```
sp1-sd pathDst(S,@D,D,P,C) :- magicSrc(@S), #link(@S,D,C),
                              P = f_init(S,D).
sp2-sd pathDst(S,@D,Z,P,C) :- pathDst(S,@Z,Z1,P1,C1),#link(@Z,D,C2),
                              C = C1 + C2, P = f_concatPath(P1,D).
sp3-sd spCost(@D,S,min<C>) :- magicDst(@D),pathDst(S,@D,Z,P,C).
sp4-sd shortestPath(S,@D,P,C) :- spCost(S,@D,C),pathDst(S,@D,Z,P,C).
```

Figure 8.2: Magic-Shortest-Path Program.

The left-recursive *Shortest-Path* program computes 1-hop paths starting from each *magicSrc* using rule sp1-sd. Rule *sp2-sd* then recursively computes new paths by following all reachable links, and stores these paths as *pathDst* tuples at each destination. Rules sp3-sd and sp4-sd then filter relevant paths based on *magicDst*, and compute the shortest paths, which can then be propagated along the shortest paths back to the source node. In fact, executing the program in this *"Top-Down" (TD)* fashion resembles a network protocol called *dynamic source routing* (DSR) [64] which we presented in Section 3.3.4 as a declarative routing example program. DSR is proposed for ad-hoc wireless environments, where the high rate of change in the network makes such targeted path discovery more efficient compared to computing all-pairs shortest paths.

Interestingly, the use of magic sets and predicate reordering reveals close connections between query optimizations and network optimizations. By specifying routing protocols in *NDlog* at a high level, we demonstrate that the two well-known protocols – one for wired networks and one for wireless – differ only in applying a standard query optimization: the order of two predicates in a single rule body. In addition, the use of magic sets allows us to do a more targeted path discovery suited in the wireless setting. Ultimately, we hope that such connections between query optimizations and network optimizations will provide a better understanding of the design space of routing protocols.

8.2 Multi-Query Optimizations

In a distributed setting, it is likely that many related queries will be concurrently executed independently by different nodes. A key requirement for scalability is the ability to share common query computations (*e.g.*, pairwise shortest paths) among a potentially large number of queries. We outline two basic strategies for multi-query sharing in this environment: *query-result caching* and *opportunistic message sharing*.

Query-Result Caching. Consider the *Magic-Shortest-Path* program where node *a* computes *shortestPath(@a,d,[a,b,d],6)* to node *d*. This cached value can be reused by all queries for destination *d* that pass through *a*, *e.g.*, the path from *e* to *d*. Currently, our implementation generates the cache internally, building a cache of all the query results (in this case *shortestPath* tuples) as they are sent back on the reverse path to the source node. Since the subpaths of shortest paths are optimal, these can also be cached as an enhancement.

Opportunistic Message Sharing. In the previous example, we considered how different nodes (src/dst) could share their work in running the *same* program logic with different constants. Sharing across *different* queries is a more difficult problem, since it is non-trivial to detect query containment in general [25]. However, we observe that in many cases, there can be correlation in the message patterns even for different queries. One example arises when different queries request "shortest" paths based on different metrics, such as latency, reliability and bandwidth; *path* tuples being propagated for these separate queries may be identical modulo the metric attribute being optimized.

A strategy that we have implemented is *opportunistic message sharing*, where multiple outgoing tuples that share common attribute values are essentially joined into one tuple if they are outbound to the same destination; they are re-partitioned at the receiving end. This achieves the effects of jointly rewriting the queries in a fashion, but on an opportunistic basis: derivations are done in this combined fashion only in cases that are spatiotemporally convenient during processing. In order to improve the odds of achieving this sharing, outbound tuples may be buffered for a time and combined in batch before being sent.

As an alternative to this opportunistic sharing at the network level, one can achieve explicit sharing at a logical level, *e.g.*, using correlated aggregate selections for pruning different paths based on a combination of metrics. For example, consider running two queries: one that computes shortest latency paths, and another that computes max-bandwidth paths. We can rewrite these as a single *NDlog* program by checking two aggregate selections, *i.e.*, only prune paths that satisfy *both* aggregate selections.

8.3 Hybrid Rewrites

Currently, rules are expressed using a left-recursive (BU) or right-recursive (TD) syntax (Section 8.1.2). Our main goal during query execution is *network efficiency* (*i.e.*, reducing the burden on the underlying network), which, typically, also implies faster query convergence. It is not difficult to see that neither BU nor TD execution is universally superior under different network/query settings. Even in the simple case of a shortest-path discovery query *shortestPath(@S,@D,P,C)* between two given nodes *(@S,@D)*, minimizing message overhead implies that our query processor should prefer a strategy that restricts execution to “sparser” regions of the network (*e.g.*, doing a TD exploration from a sparsely-connected source @*S*).

We argue that *cost-based* query optimization techniques are needed to guarantee effective query execution plans. While such techniques have long been studied in the context of relational database systems, optimizing distributed recursive queries for network efficiency raises several novel challenges. In the remainder of this section, we briefly discuss some of our preliminary ideas in this area and their ties with work in network protocols.

The Neighborhood Function Statistic. As with traditional query optimization, cost-based techniques must rely on appropriate *statistics* for the underlying execution environment that can drive the optimizer's choices. One such key statistic for network efficiency is the *local neighborhood density* $N()$. Formally, $N(X,r)$ is the number of distinct network nodes within r hops of node X. The neighborhood function is a natural generalization of the size of the transitive closure (*i.e.*, reachability set) of a node, that can be estimated locally (*e.g.*, through other recursive queries running in the background/periodically). $N(X,r)$ can also be efficiently *approximated* through approximate-counting techniques using small (log-size) messages [108]. To see the relevance of $N()$ for our query-optimization problem, consider our example *shortestPath(@s,@d,P,C)* query, and let $\texttt{dist}(s,d)$ denote the distance of s, d in the network. A TD search would explore the network starting from node s, and (modulo network batching) result in a total of $N(s,\texttt{dist}(s,d))$ messages (since it reaches all nodes within a radius of $\texttt{dist}(s,d)$ from s). Note that each node only forwards the query message once, even though it may receive it along multiple paths. Similarly, the cost for a BU query execution is $N(d,\texttt{dist}(s,d))$. However, neither of these strategies is necessarily optimal in terms of message cost. The optimal strategy is actually a *hybrid scheme* that "splits" the search radius dist(s,d) between s and d to minimize the overall messages; that is, it first finds r_s and r_d such that:

$$(r_s,r_d) \;=\; \arg \min_{r_s+r_d=\texttt{dist}(s,d)} \{\, N(s,r_s)+N(d,r_d)\,\},$$

and then runs concurrent TD and BU searches from nodes s and d (with radii r_s and r_d, respectively). At the end of this process, both the TD and the BU search have intersected in at least one network node, which can easily assemble the shortest (s,d) path. While the above optimization problem is trivially solvable in $O(\texttt{dist}(s,d))$ time, generalizing this hybrid-rewrite scheme to the case of multiple sources and destinations raises difficult algorithmic challenges. And, of course, adapting such cost-based optimization algorithms to work in the distributed, dynamic setting poses system challenges. Finally, note that neighborhood-function information can also provide a valuable indicator for the utility of a node as a result cache (Section 8.2) during query processing.

Adaptive Network Routing Protocols. As further illustrations on the close connection between networking routing and query optimizations, we note that the networking literature has considered adaptive routing protocols that strongly resemble our use of hybrid rewrites; hence, we believe this is an important area for future investigation and generalization. One interesting example is the class of *Zone-Routing Protocols* (ZRP) [55]. A ZRP algorithm works by each node precomputing *k-hop-radius* shortest paths to neighboring nodes (in its "zone") using a BU strategy. Then, a shortest-path route from a source to destination is computed in a TD fashion, using essentially the *Magic-Shortest-Path* program described above, utilizing any precomputed shortest paths along the way. Each node sets its zone radius k adaptively based on the density and rate of change of links in its neighborhood; in fact, recent work [101] on adjusting the zone radius for ZRP-like routing uses exactly the neighborhood-function statistic.

8.4 Evaluation of Optimizations

In this section, we examine the effectiveness of the optimizations that are proposed in this chapter. We base our workload primarily on declarative routing protocols, and measure four variants of the same *Shortest-Path* program, differing in the link metric each seeks to minimize. Our experimental setup is similar to Section 7.1, where we executed the *Shortest-Path* program on an overlay network in which each node has four neighbors. In addition, for each neighbor link, we generate additional metrics that include reliability, and a randomly generated value. Note that our reliability metric for each link is synthetically generated to be correlated with latency.

On all our graphs, we label these queries by their link metric: *Hop-Count*, *Latency*, *Reliability* and *Random*, respectively. Recall from Section 7.1.2 that *Random* serves as our stress case: we expect it to have the worst performance among the different metrics. This is due to aggregate selections being less effective when the aggregate metric is uncorrelated with the network latency.

8.4.1 Aggregate Selections

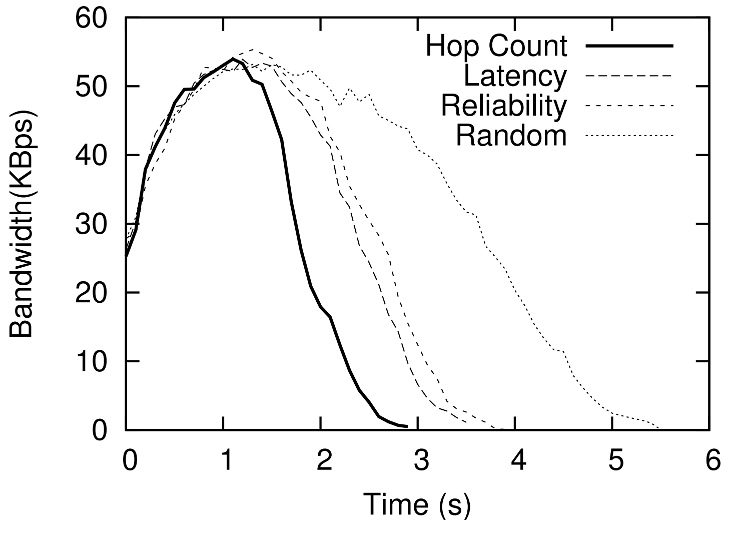

Figure 8.3: Per-node Bandwidth (KBps) with Aggregate Selections.

We first investigate the effectiveness of aggregate selections for different queries. Fig-

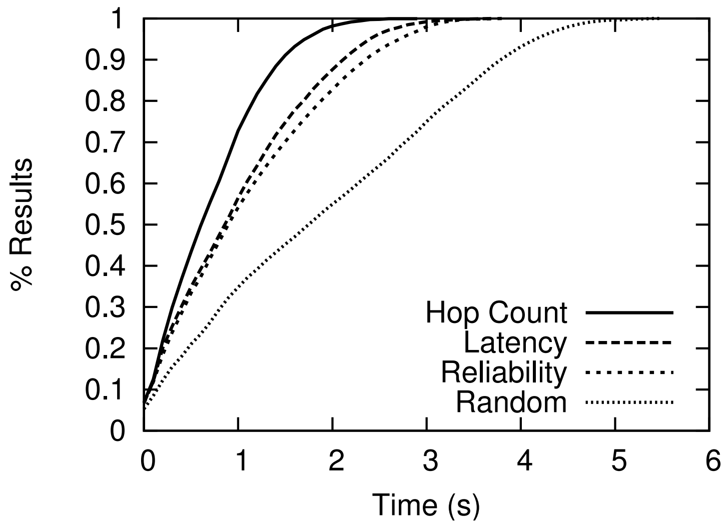

Figure 8.4: Results over time (seconds) with Aggregate Selections.

ure 8.3 shows the per-node bandwidth usage against time for the *Shortest-Path* program on all four metrics. Figure 8.4 shows the percentage of eventual best paths completed against time. Our results show that *Hop-Count* has the fastest convergence time of 2.9 seconds, followed by *Latency* and *Reliability* in 3.5 seconds and 3.9 seconds respectively. *Random* has the worst convergence time of 5.5 seconds.

During program execution, the communication overhead incurred by all four queries shows a similar trend (Figure 8.3). Initially, the communication overhead increases as more and more paths (of increasing length) are derived. After it peaks at around $53KBps$ per-node, the communication overhead decreases, as fewer and fewer optimal paths are left to be derived. In terms of aggregate communication overhead, *Random* incurs the most overhead (18.2 MB), while *Hop-Count*, *Latency* and *Reliability* use 9.1 MB, 12.0 MB and 12.8 MB, respectively. The relatively poor performance of *Random* is due to the lack of correlation between the metric and network latency, leading to a greater tendency for out-of-order arrival of path tuples that results in less effective use of aggregate selection, translating to more messaging overhead and delays.

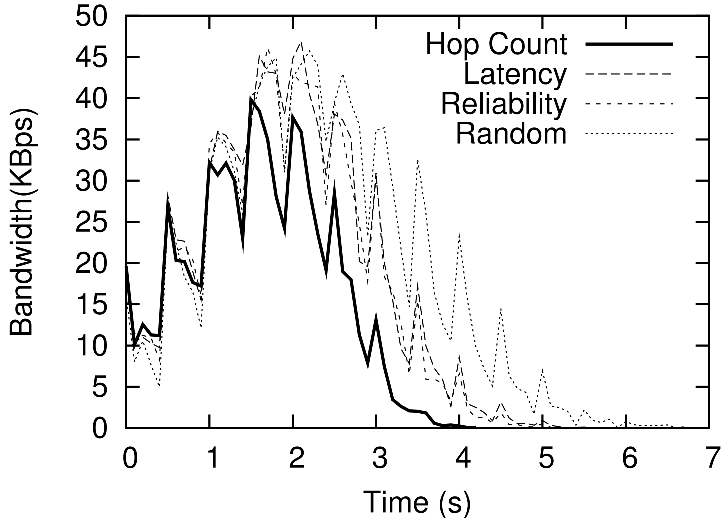

Figure 8.5: Per-node Bandwidth (KBps) with periodic aggregate selections.

The results in Figures 8.5 and 8.6 illustrate the effectiveness of the *periodic aggregate selections* approach, as described in Section 8.1.1, where the wait period is set to 500 ms. In particular, this approach reduces the bandwidth usage of *Hop-Count*, *Latency*, *Reliability* and *Random* by 19%, 15%, 23% and 34%, respectively. *Random* shows the greatest reduction in communication overhead, demonstrating the effectiveness of this technique for improving the performance of queries on metrics that are uncorrelated with network delay.

8.4.2 Magic Sets and Predicate Reordering

Next, we study the effectiveness of combining the use of magic sets and predicate reordering for lowering communication overhead when the requested shortest paths are constrained by randomly chosen sources and destinations. Our workload consists of queries

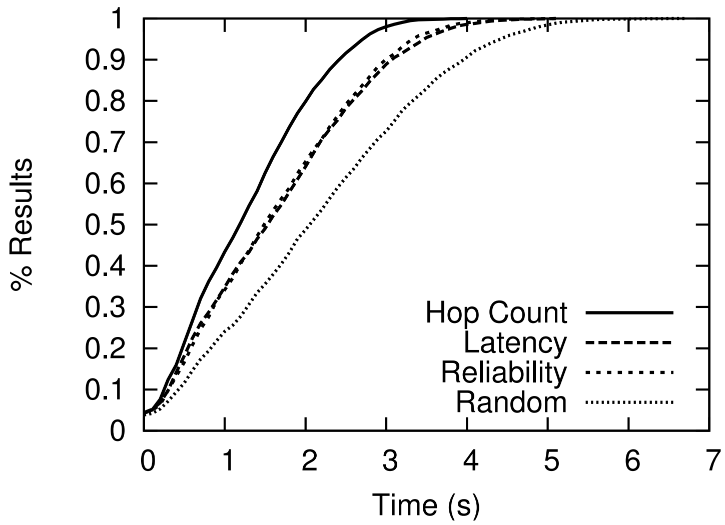

Figure 8.6: Results over Time (seconds) with periodic aggregate selections.

that request source-to-destination paths based on the *Hop-Count* metric. For each query, we execute the *Magic-Shortest-Path* program (Section 8.1.2).

Figure 8.7 shows the aggregate communication overhead as the number of queries increases. The *No-MS* line represents our baseline, and shows the communication overhead in the absence of rewrites (this essentially reduces to computing all-pairs least-hop-count). The *MS* line shows the communication overhead when running the program optimized with magic sets, but without any sharing across queries. When there are few queries, the communication overhead of *MS* is significantly lower than that of *NO-MS*. As the number of queries increases, the communication overhead of *MS* increases linearly, exceeding *No-MS* after 170 queries.

In addition, Figure 8.7 also illustrates the effectiveness of caching (Section 8.2). The *MSC* line shows the aggregate communication overhead for magic sets with caching. For fewer than 170 queries, there is some overhead associated with caching. This is due to false positive cache hits, where a cache result does not contribute to computing the even-

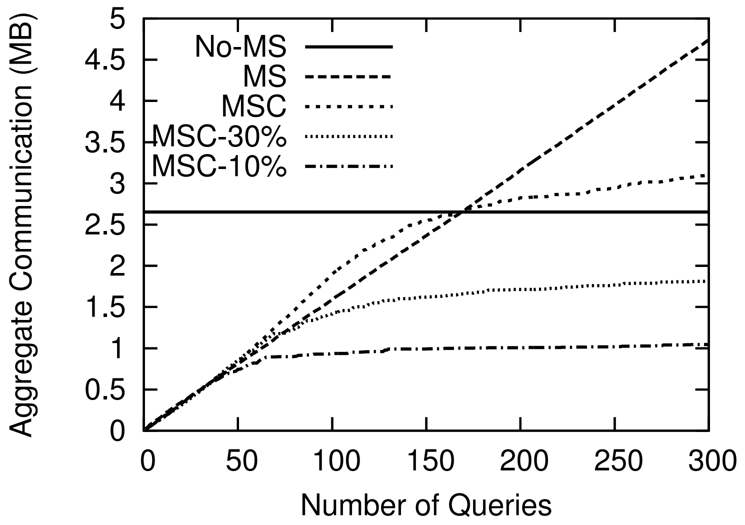

Figure 8.7: Aggregate communication overhead (MB) with and without magic sets and caching.

tual shortest path. However, as the number of queries increases, the overall cache hit rate improves, resulting in a dramatic reduction of bandwidth. When limiting the choice of destination nodes to 30% (*MSC-30%*) and 10% (*MSC-10%*), the communication overhead levels of at 1.8 MB, and 1 MB, respectively. The smaller the set of requested destinations, the higher the cache hit rate, and the greater the opportunity for sharing across different queries. e

8.4.3 Opportunistic Message Sharing

We study the impact of performing opportunistic message sharing across concurrent queries that have some correlation in the messages being sent. Figure 8.8 shows per-node bandwidth usage for running the queries on different metrics concurrently. To facilitate sharing, we delay each outbound tuple by 500 ms in anticipation of possible sharing opportunities. The *Latency*, *Reliability* and *Random* lines show the bandwidth usage of each query in-

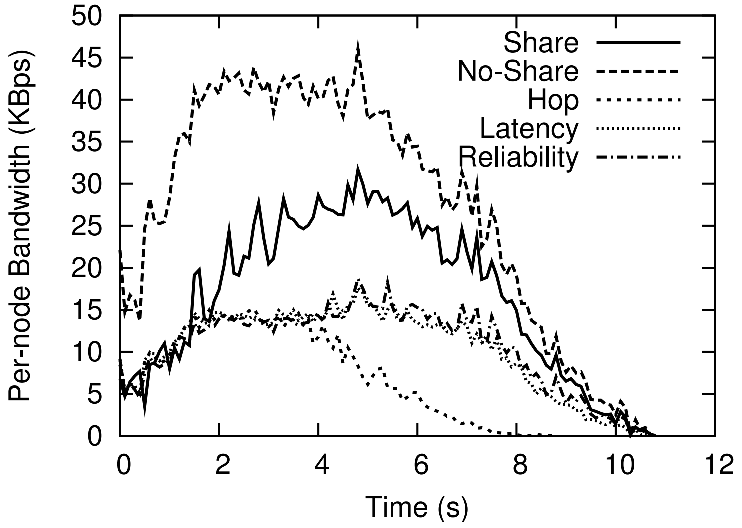

Figure 8.8: Per-node Bandwidth (KBps) for message sharing (300 ms delay).

dividually. The *No-Share* line shows the total aggregate bandwidth of these three queries without sharing. The *Share* line shows the aggregate bandwidth usage with sharing. Our results clearly demonstrate the potential effectiveness of message sharing, which reduces the peak of the per-node communication overhead from 46 KBps to 31 KBps, and the total communication overhead by 39%.

8.4.4 Summary of Optimizations

We summarize our evaluation as follows:

1. The aggregate selections optimization indeed reduces communication overhead. Using *periodic aggregate selections* reduces this overhead further.

2. The use of magic sets and predicate reordering reduces communication overhead when only a limited number of paths are queried.

3. Multi-query sharing techniques such as reusing previously computed results and opportunistic result caching demonstrate the potential to reduce communication overhead when there are several concurrent queries.

8.5 Summary

In this chapter, we applied a variety of query optimizations to declarative networks. We explored the use of traditional query optimizations and proposed new optimizations motivated by the distributed setting. We demonstrated that declarative networks are amenable to automatic optimizations, and showed that many of these optimizations can improve the performance of declarative networks substantially. In addition, we validated the effectiveness of several of our optimization techniques on the Emulab testbed.

Interestingly, we revealed surprising relationships between network optimizations and query optimizations, *e.g.*, a wired protocol can be translated to a wireless protocol by applying the standard database optimizations of magic sets rewrite and predicate reordering. This suggests that these protocols are more similar than the are often made out to be. By drawing the connections between network optimizations and query optimizations, we set the groundwork for a series of more focused investigations in the future. We revisit these issues as future work in Section 10.3.

Chapter 9

Related Work

In this chapter, we summarize related work in both the database and networking domain, focusing on deductive databases for processing recursive queries, distributed query processors, extensible networks and network specification languages. In addition, there is also a wide variety of related work in dataflow architectures such as Click [65] which we have reviewed and compared with our system in Chapter 4.

9.1 Deductive Databases

A *deductive database* system is a database system which can make deductions (*i.e.*, infer additional rules or facts) based on rules and facts stored in the database. Datalog is a query and rule language for deductive databases that syntactically is a subset of Prolog [32]. Unlike Prolog, Datalog utilizes a bottom-up (or forward-chaining) evaluation strategy that guarantees termination, and also permits set operations. This comes at the potential expense of redundant computations which can be avoided with query optimizations such as the magic sets rewrite [18; 16] described in Chapter 8.

One of the key features of Datalog is its support for recursive queries [14]. A primary use of deductive database systems is hence for supporting queries over graphs that themselves exhibit recursive properties. The database literature has a rich tradition of research on recursive query languages and processing. This work has influenced commercial database systems to a certain extent. However, recursion is still considered an esoteric feature by most practitioners, and research in the area has had limited practical impact. Even within the database research community, there is longstanding controversy over the practical relevance of recursive queries, going back at least to the Laguna Beach Report [21], and continuing into relatively recent textbooks [117].

In addition to our work, there has been recent renewed enthusiasm for applications of recursive queries. There are other contemporary examples from outside the traditional database research literature, including software analysis [127], trust management [17] and diagnosis of distributed systems [5] and network security analysis [129]. Our concept of *link-restricted* rules is similar in spirit to *d3log* [63], a query language based on Datalog proposed for dynamic site discovery along web topologies.

In terms of distributed systems, the closest analog is the recent work by Abiteboul *et al.* [5]. They adapt the QSQ [71] technique to a distributed domain in order to diagnose distributed systems. An important limitation of their approach is that they do not consider partitioning of relations across sites as we do; they assume each relation is stored in its entirety in one network location. Further, they assume full connectivity and do not consider updates concurrent with query processing.

Much research in the parallel execution of recursive queries [24] has focused on high throughput within a cluster. In contrast, our strategies and optimizations are geared towards bandwidth efficiency and fast convergence in a distributed setting. Instead of hash-based partitioning schemes that assume full connectivity among nodes, we are required to perform query execution only along physical network links, and to deal with network changes during query execution. There is also previous empirical work on the performance of parallel pipelined execution of recursive queries [111]. Our results extend that work by providing new, provably correct pipelining variants of semi-naïve evaluation.

9.2 Internet-Scale Query Processing

There has been substantial work in the area of distributed [89; 116] and parallel database systems [39]. While parallelism per se is not an explicit motivation of our work, algorithms for parallel query processing form one natural starting point for systems that process queries on multiple machines. For example, our *localization rewrite* described in Chapter 5.2 builds upon previous work on data repartitioning during joins in systems like Gamma [38] and Volcano [51]. Of greater relevance to our work on declarative networking is research on *Internet-scale* query processing, where the focus is on building distributed querying facilities over data on the Internet. Examples of these systems include PIER [60; 59; 78; 81], Iris Net [49], Astrolabe [104] etc.

Of these Internet-scale query processing systems, the architecture of P2 has been most inspired by the PIER system. In Chapter 4, we describe the similar ties in terms of the dataflow framework of PIER and P2. While the focus of PIER has been on supporting SQL-like queries, PIER also supports recursive dataflows that have been used for building distributed crawlers for the web [82] and overlay networks [77].

One major distinction with PIER is the fact that PIER couples its design tightly with the use of a distributed hash table [103; 114; 107; 131; 102] as its basic common substrate, which is then employed to instantiate query-specific overlay networks such as aggregation trees. In contrast, P2 simply uses whatever underlying network is present, and each node can be configured with a relatively small set of “base facts” (such as addresses of a few nearby neighbors). Knowledge of the rest of the network is then built up in the declarative domain. It is possible to construct a DHT using P2 – indeed, one of our examples in this dissertation is a version of Chord – but P2 in no way requires a DHT to be present, nor relies on the assumptions a DHT typically exploits (such as full-mesh connectivity between nodes, and lack of explicit control over node and data placement). Interestedly, the generality of P2 means that it is possible to reimplement PIER entirely by expressing relational queries over a declarative version of Chord specified using the P2 system.

9.3 Extensible Networks

There have been many recent proposals for increasing the flexibility of routing in the context of the Internet. Proposed solutions include enabling end-hosts to choose paths at the AS level [130; 69], separating routing from the forwarding infrastructure [68; 44], centralizing some of the routing decisions [44], and building extensible routers such as XORP [57; 56]. Our proposal is mostly complementary to these efforts. The increased flexibility provided by a declarative interface can enhance the usability and programmability of these systems. Our proposal is also orthogonal to the separation of the control plane and the data plane. As discussed in Chapter 3, our system can be deployed fully centralized, distributed or partially centralized for supporting Internet-scale routing.

Several type-safe languages have been proposed to improve the security and robustness of active networks. Three examples are PLAN [58], PLAN-P [121] and SafetyNet [109]. Compared to these languages, Datalog is particularly attractive because of its strong theoretical foundations, the fact that it is a side-effect-free language sandboxed within a query engine, and its elegance in expressing routing protocols in a compact way. Unlike previous proposals, as a declarative query language, Datalog is also amenable to automatic query optimization techniques from the database literature.

9.4 Network Specification Languages

In recent years, there have been efforts at high-level specifications of network protocols [52; 42]. These specifications aim at verifying correctness properties of Internet-routing protocols. They are less general than our work on declarative networking. One can conceivably implement some of these specifications using the P2 system, as a way of bridging specifications and implementation.

In the past, distributed systems have typically been characterized in one of two ways. The *protocol-centric* approach favored by Macedon [105] traces its roots to event languages [45; 122] that specify overlay execution via automata for event and message handling. This style emphasizes the dynamics of the overlay and its maintenance, but makes it difficult to determine the overlay's coarse structure and invariant properties. The alternative is a *structure-centric* approach, whose roots can be traced to the specification of parallel interconnection networks [72]. This style, which has influenced the literature on distributed hash tables (DHTs), specifies overlays by focusing on a network graph structure (hypercube, torus, de Bruijn graph, small-world graph, etc.), whose invariant properties must be maintained via asynchronous messaging. Unfortunately, graph-theoretic descriptions tend to be expressed at a high level in natural language, and often gloss over details of the actual runtime messaging. As a result, implementing structure-centric overlays often requires a fair bit of engineering [103; 34], and different implementations of the same overlay can vary significantly in their actual execution.

P2 spans the two approaches above, and expands upon them in a way that is particularly attractive for overlay specification and runtime. The interface of P2 is closer in spirit to the structure-centric approach, in that it encourages the specification of overlays as logical structures with invariants. However, it also automatically compiles this specification to a dataflow program for managing asynchronous messages, which looks closer to the protocol-centric approach. We believe P2 improves upon previous overlay specification work in either camp, by providing a machine-interpretable description language based on relations among node states in the network, and by using a dataflow runtime model instead of automaton-based protocols.

The combination of a declarative language and a dataflow runtime forms a powerful and surprisingly natural environment for overlay specification and runtime. The obvious alternative to our approach is the automaton approach used in traditional protocol specifications and implementations, and in the Macedon overlay toolkit. Relative to automata, logical specifications and dataflow graphs have a number of software engineering advantages:

- **Scoping:** In principle, automata must handle any possible event (message) in each state. While automata can in principle be nested or encapsulated as a matter of design discipline, the potentially arbitrary interplay between states and events leads to relatively few design constraints, making automata difficult both to specify correctly, and to understand once specified. By contrast, in a dataflow diagram compiled from an *NDlog* program, the inputs to an element are by definition coming from other specific elements whose behavior is well specified. This constrains what needs to be handled in each element implementation, aiding in both specification and comprehension.

- **Typing:** Similarly, the events or messages handled in automata are of any type possible in the system. In dataflow diagrams, all tuples that pass along an edge share the same schema, hence a dataflow element implementation need only worry about a stream of similar, well-formed tuples.

- **Encapsulation and Reuse:** Because automata interrelate possible events and states, they are difficult to reuse in other contexts that may involve different sets of events, or additional states with different interactions. By contrast, subsets of rules in *NDlog* programs can be easily extracted and reused in other programs. Moreover, even the compiled dataflow diagrams often have discrete subgraphs that are clearly reusable: a dataflow subgraph typically has a few well-specified inputs (incoming edges) and outputs (outgoing edges), and in many cases has easily interpretable behavior. This admits the possibility of allowing incoming programs to opportunistically "jump on" to existing dataflows, in the spirit of adaptive stream query engines like TelegraphCQ [27].

Chapter 10

Conclusion

In this chapter, we conclude the dissertation by (1) summarizing our contributions, (2) surveying some recent use cases of the P2 system, and (3) proposing several directions for future work.

10.1 Summary of Contributions

Our work on declarative networking has two high-level goals. First, through the use of a declarative language, we aim to greatly simplify the process of specifying, implementing, deploying and evolving a network design. Second, we aim to address the challenge of designing flexible, secure and efficient network architectures that can achieve a better balance between flexibility and security compared to existing solutions. We summarize our key contributions as follows:

- We formally define the *Network Datalog* (*NDlog*) language for declarative networking. The *NDlog* language has its roots in logic programming and recursive query languages [98]. It is based on the observation that recursive queries are a natural way to express network protocols that themselves exhibit recursive properties. *NDlog* builds upon Datalog, a traditional recursive query language, to enable *distributed* and *soft-state* computations with *restricted communication* based on the underlying physical connectivity, all of which are essential in the network setting.

We show that the *NDlog* language is a compact and natural way of expressing a variety of routing protocols [79; 80] and overlay networks [76], often resulting in orders of magnitude savings in code sizes. *NDlog* programs are compiled into dataflow execution plans which are then executed using a distributed query engine in the P2 system to implement the network protocols. In addition to being concise, we further show that *NDlog* programs are amenable to query optimizations and static analysis, making it an attractive language for building safe, extensible network architectures.

- Second, to validate the design of *NDlog*, we present our implementation of P2 [2], which is a full-fledged declarative networking system with a dataflow engine inspired by the Click modular router [65]. The P2 system takes as input *NDlog* programs, compiles them into distributed execution plans that are then executed by a distributed dataflow engine to implement the network protocols. We experimentally evaluate the P2 system on hundreds of distributed machines running a wide variety of different protocols, including traditional routing protocols as well as complex overlays such as distributed hash tables. Our experiments validate that our declarative specifications result in the expected network properties in terms of topology and performance. In addition, our Emulab experiments demonstrate that query optimizations are effective in improving the performance of declarative networks.

- Based on our experiences in implementing declarative networks, we make fundamental contributions that further advance the state of the art in recursive query processing. We explore a wide variety of database research issues that are important for the practical realization of declarative networks. These include reasoning about correctness of query results based on the well-known distributed systems notion of "eventual consistency", pipelined execution of distributed recursive queries to deal with asynchrony in networks, incremental view materialization over dynamic and soft-state data, and query optimizations. Interestingly, we show that the *NDlog* specifications for two well-known protocols – one for wired networks and one for wireless – differ only in applying a standard query optimization: the order of two predicates in a single rule body. We believe that the connections we have drawn between automatic query optimizations and network protocol design can both provide a better understanding of the design space of routing protocols, and also open up significant new avenues of research in both query optimization and protocol design.

10.2 Broad Use of Declarative Networking

One of the metrics of long-term success for this work will be the adoption of declarative networking ideas, both within academia and industry. Our short-term goal is to promote declarative networking as a tool for rapid prototyping and experimentation with new network designs. In addition, a variety of practical Internet-scale data intensive applications can be built to leverage declarative networks. These applications include network monitoring [59], information retrieval [78; 73], distributed web and network crawling [60; 82] and distributed replication protocols [19; 120]. In the long term, one ambitious goal is that declarative networks can serve as a core for future Internet designs, and help fuel the next generation of "clean-slate" Internet architectures.

The P2 implementation currently runs over Linux and Mac OS X and consists of around 50,000 lines of C++ and python, plus 3rd-party support libraries. The system has been available for download at http://p2.cs.berkeley.edu since February 2006. The implementation includes a runtime system that allows for incremental rule planning and query dissemination. In addition to running on local clusters and Emulab, we have also deployed the P2 system on Planetlab [94].

In making the P2 system publicly available, one of our objectives is to explore more use cases of the system that will enable us to better understand the strengths and limitations of *NDlog*. This will also enable us to identify common constructs that can be factored out of particular network specifications and shared. Following our code release, the P2 system has been used in research projects at various institutions (*e.g.*, Cambridge University, Harvard University, Intel Research at Berkeley, KAIST, Max Planck Institute, Rice University, University of California-Berkeley and University of Texas-Austin). We survey some recent uses of the system:

- Singh et al. [112] demonstrated that the P2 system not only can implement declarative networks, but also conveniently provide powerful distributed *debugging* facilities for these networks. Using runtime system invariants that are expressed as additional *NDlog* rules, they demonstrate logging, monitoring, causality and debugging facilities that they built on top of the P2 system. They used the P2 system to implement a range of on-line distributed diagnosis tools that range from simple, local state assertions to sophisticated global property detectors on consistent snapshots.

- Condie et al. [123] proposed a highly decomposable and componentized transport layer for the networking stack of the P2 system. This reconfigurable networking stack is implemented at the dataflow level of the P2 system, and can be utilized by declarative networks to adapt at run-time to application-specific requirements.

- As a class project, students from Harvard University implemented the Paxos consensus protocol [20] in 42 *NDlog* rules using the P2 system.

- There are several ongoing projects that are either using the P2 system or applying declarative networking concepts in their own systems. These projects range from implementing the Pastry [107] DHT with induced churn [124], prototyping replication algorithms (*e.g.*, PRACTI [19]), programming sensor networks [29] and implementing parallel dataflow processing in clusters [35].

In order to achieve our long-term goal of having an impact on future Internet design, an important future step is to encourage the adoption of declarative networking by ISPs. This will involve integrating distributed query processors like P2 with commercial routers, and ensuring that declarative networks can interoperate with existing intra-domain and inter-domain routing protocols. Once deployed in routers, these query processors can concurrently be used for extensible routing and network monitoring.

10.3 Research Directions

Broadly, we believe that our work can impact the networking and database communities in the following ways. For the networking community, our work has the potential to dramatically alter the way networking protocols are designed, implemented and verified. For the database community, this dissertation can be a first step towards not only rekindling interest in recursive query research, but also generating new insights into wide area data management, communication, and synergies available by intertwining the two within a single declarative framework. In general, we are optimistic that this research can lead to significant results in this domain, in terms both of theoretical work and systems challenges.

In this section, we highlight some future directions beyond what has been presented in this thesis.

10.3.1 Static and Runtime Analysis of Networks

An important potential of declarative networking that we have not fully explored is *checkability*: the promise of static program checks for desirable network protocol properties (*e.g.*, convergence and stability) to ensure program safety in extensible networks.

Static checks are unlikely to solve all problems. However in cases where static checking is not able to provide a sound and complete analysis, there is hope that runtime checks [112] could be *automatically synthesized* by a compiler, and added to the program as additional datalog rules to ensure desirable properties. Together with these runtime checks, declarative networking can potentially serve as an integrated system for network specification, implementation and verification.

In the short term, one immediate challenge is to study the applicability of static analysis techniques from the Datalog literature [67] on query safety, and extend these analysis techniques to handle soft-state data and rules, distributed computations and link-restricted communication. In addition, there are open questions as to whether these techniques can be easily integrated with recent attempts at building verifiable network specifications [52; 42]. An interesting first step will be to build a compiler that translates these verifiable network specifications to *NDlog* programs for execution using P2. A longer term challenge involves incorporating verification techniques from the formal methods community [122; 70] and software model checking approaches [41] into analyzing declarative networks.

10.3.2 Processing NDlog Programs with Negation

In this dissertation, we do not consider negated predicates in the *NDlog* language. The main reason lies in the difficulties in ensuring correct semantics if negation [61] is incorporated into our model and implementation. Here, we provide high-level intuitions on the difficulties of handling negation in *NDlog* and possible approaches.

In the context of traditional deductive databases, there are several proposals on defining the “right” meaning to rules with negated predicates. The straightforward case involves Datalog programs with *stratified negation*. These programs do not have cyclical (recursive) dependencies involving negated predicates. The standard technique for processing these rules involve the use of *stratification*, where the program is broken up into different *strata* or layers based on the dependency graph of the predicates across all the rules. The program is then executed one stratum at a time.

Unfortunately, stratified negation is overly restrictive, and in practice, many Datalog programs contain cyclical dependencies involving negated predicates. Over the years, there have been several different proposals on coming up with the appropriate model for these programs. These models include *local stratification*, *well-founded negation*, *stable-model negation* and *modularly stratified negation* (See [61] for a good overview on the different models). Despite active research by the deductive database community in the past, picking the appropriate model remains an open challenge.

In the distributed setting involving programs with stratified negation, there are challenges involved in efficiently implementing stratified negation. A prohibitively expensive solution involves achieving consensus among multiple nodes executing the same stratum before moving to a new stratum. The practical alternative involves incorporating pipelined execution, where results are “eagerly” computed across stratum, and incrementally updated to deal with cases when there is a “late” counterexample to a negated subgoal. The use of pipelined execution of *NDlog* programs with negation raises additional challenges due to the presence of dynamic data, and can be a rich area of future exploration. It is also interesting to come up with compelling examples of declarative networks where negation would be useful.

10.3.3 Declarative Language and System for Secure Networking

Beyond static and runtime analysis of *NDlog* programs, there still remains the possibility of other security challenges, including malicious routers and denial of service attacks. In response to these attacks, independent of our work, the security community has proposed several mechanisms, and these have been formalized in several declarative logic-based security languages such as Binder [37], SD3 [62], D1LP [74] and SecPAL [86] for access control [3] and trust management [23].

Interestingly, Binder and *NDlog* extend traditional Datalog in similar ways: by supporting the notion of location (or *context*) to identify nodes (or *components*) in distributed systems. This suggests the possibility of unifying these languages to create an integrated system, exploiting good language features, execution and optimizations. From a practical standpoint, this integration has several benefits, including ease of management, one fewer language to learn, one fewer set of optimizations, finer-grain control over the interaction between security and network protocol, and the possibility of doing analysis and optimizations across levels. This integrated system could have broad applicability, ranging from building secured networks [26], secure distributed information sharing [84], to enforcing access control policies on extensible testbeds such as GENI [48].

This further suggests that we may be able to dispense with much of the special machinery proposed for access control, and instead process security policies using distributed database engines. This allows us to leverage well-studied query processing and optimization techniques. Interestingly, it has been shown previously [4] that Binder is similar to data integration languages such as Tsimmis [28] proposed by the database community, further indicating that ideas and methods from the database community are directly applicable to secure networking.

10.3.4 Database Techniques for Network Optimizations

In Chapter 8, we identified a few well-known query optimization techniques and showed how they can be used to generate efficient protocols. While the application of query optimizations automatically achieves some well-known optimizations for routing protocols, it will be interesting to see how they can help inform new routing protocol designs, especially when applied to more complex networks.

One promising direction stems from our surprising observation in Chapter 8 on the synergies between query optimization and network routing: a wired protocol can be translated to a wireless protocol by applying the standard database optimizations of magic sets rewrite and predicate reordering. This suggests that these protocols are more similar than they are often perceived to be. It suggests that protocol selection can be achieved by cost estimation a la traditional database query optimization rather than rules of thumb about "what kind of network" is in place. In fact, many future architectures for the Internet envision a very heterogeneous network that combines wired infrastructure with wireless "clouds" at the edges in a more seamless way than we have today. In those situations, it would be nice for the infrastructure to choose efficient protocols based on cost estimation. It also suggests that sites could make the decision about whether to apply this optimization locally (or within a neighborhood), allowing for the hybridization of the protocol within the network.

Building upon our techniques for cost-based optimizations proposed in Section 8.3, it would be interesting to study the possibility of using a number of other potential optimization strategies: random walks driven by statistics on graph expansion; cost-based [110] decisions on the application of magic sets; adaptive [10] query processing techniques to react to network dynamism; and multi-query optimizations motivated by more complex overlay networks. The ultimate goal is to fully understand the synergies between query optimization and network routing, with the hope of informing new protocol designs that can improve upon the performance of existing networks.

In Section 8.2, we highlighted some techniques for sharing across multiple declarative routing programs. In future, these sharing techniques can be extended for more complex overlay networks. Sharing of declarative overlays is intriguing not only in terms of code reuse, but also for the possibility that multiple overlays can execute simultaneously, sharing state, communication, and computation by sharing dataflow subgraphs. Sharing between multiple declarative overlays can allow a single application to achieve different performance goals for different tasks, by efficiently deploying multiple overlay variants simultaneously. For example, a peer-to-peer file sharing network that combines search with parallel download might choose to construct two different overlays for these very different tasks. These overlays might fruitfully share rules for liveness checking, latency and bandwidth estimation, etc. Runtime sharing across overlays can also allow separately deployed systems to co-exist efficiently within a shared network infrastructure; this might become important if overlays emerge as a prevalent usage model for the Internet. The naïve approach for sharing is to do so explicitly at the *NDlog* level, by sharing standard libraries of rule specifications or caching previously computed results. However, we hope to apply multiquery optimization techniques from the database literature to identify further sharing opportunities automatically within a sophisticated query optimizer.

10.3.5 Application-Aware Extensible Networks

Traditional distributed databases, or even recent proposals on Internet-scale distributed query engines, have all made the *fixed-protocol* assumption. While the set of nodes participating in these systems may change over time, the protocols used to organize nodes and route data remain unchanged. This has constrained the way data is placed and queried. One example of this constraint is in P2P search, where the available indexing and search techniques are limited by the underlying network topology [78].

One promising use of declarative networks is in building *application-aware extensible* *networks* that can be declaratively specified, and then reconfigured at runtime based on application requirements and network conditions. In parallel, several optimization strategies in distributed systems have been recently studied as cost-aware resource management frameworks [100; 88] by both the networking and database communities. It will be interesting and useful to explore incorporating some of these optimizations into a cost-based optimizer for declarative networks. These application-aware extensible networks will have wide applicability, in the domain of data intensive wide-area applications including P2P search, network monitoring, data integration and content-based routing systems.

Related to this effort is the language challenge of developing useful *network abstractions* (or building blocks) that make it easier for network designers to specify networks. For example, it has been shown that one can abstractly reason about distributed hash tables in terms of their components [66]: geometry, route selection and neighbor selection policies. It is an interesting challenge to extend *NDlog* to support such abstractions in an encapsulated way. The cost-based optimizer can then individually and jointly optimize each component based on application requirements.

10.4 Closing

In the coming years, we are entering a period of significant flux in network services, protocols and architecture. Extensible networks will be required so that a multiplicity of unforeseeable new applications and services can be handled easily in a unified manner.

This dissertation presents a declarative language and system that can greatly simplify the process of specifying, implementing, deploying and evolving a network. We demonstrate that *NDlog* programs are a natural and compact way of expressing a variety of well-known network protocols. We further show that *NDlog* programs can be compiled into distributed dataflows and executed using the P2 system to correctly implement the network protocols. Our experience with the P2 system suggests that implementing this new technology is a manageable and viable undertaking for any extensible network architecture.

Bibliography

[1] Foreign Language Functions in LDL++. http://www.cs.ucla.edu/ldl/tutorial/foreign.html.

[2] P2: Declarative Networking. http://p2.cs.berkeley.edu.

[3] ABADI, M. Logic in Access Control. In *Symposium on Logic in Computer Science* (2003).

[4] ABADI, M. On Access Control, Data Integration and Their Languages. *Computer Systems: Theory, Technology and Applications, A Tribute to Roger Needham Springer-Verlag* (2004), 9–14.

[5] ABITEBOUL, S., ABRAMS, Z., HAAR, S., AND MILO, T. Diagnosis of Asynchronous Discrete Event Systems - Datalog to the Rescue! In *ACM Symposium on Principles of Database Systems* (2005).

[6] ABITEBOUL, S., HULL, R., AND VIANU, V. *Foundations of Databases*. Addison-Wesley, 1995.

[7] AKAMAI. Akamai Content Distribution Network. http://www.akamai.com.

[8] ANDERSON, T., PETERSON, L., SHENKER, S., AND TUNER, J. Overcoming barriers to disruptive innovation in networking. Report of NSF Workshop, Jan. 2005.

[9] ARNI, F., ONG, K., TSUR, S., WANG, H., AND ZANIOLO, C. The Deductive Database System LDL++. *The Theory and Practice of Logic Programming 2* (2003), 61–94.

[10] AVNUR, R., AND HELLERSTEIN, J. M. Eddies: Continuously adaptive query processing. In *Proceedings of ACM SIGMOD International Conference on Management of Data* (2000).

[11] BALBIN, I., AND RAMAMOHANARAO, K. "a generalization of the differential approach to recursive query evaluation". *Journal of Logic Programming 4*, 3 (1987).

[12] BALBIN, I., AND RAMAMOHANARAO, K. A Generalization of the Differential Approach to Recursive Query Evaluation. *Journal of Logic Prog, 4(3):259–262* (1987).

[13] BALLARDIE, T., FRANCIS, P., AND CROWCROFT, J. Core Based Trees (CBT): An Architecture for Scalable Inter-Domain Multicast Routing. In *Proceedings of ACM SIGCOMM Conference on Data Communication* (2003).

[14] BANCHILLON, F., AND RAMAKRISHNAN, R. An amateur's introduction to recursive query processing strategies. In *Proceedings of ACM SIGMOD International Conference on Management of Data* (1986).

[15] BANCILHON, F. Naive Evaluation of Recursively Defined Relations. *On Knowledge Base Management Systems: Integrating AI and DB Technologies* (1986).

[16] BANCILHON, F., MAIER, D., SAGIV, Y., AND ULLMAN, J. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proceedings of ACM SIGMOD International Conference on Management of Data* (1986).

[17] BECKER, M. Y., AND SEWELL, P. Cassandra: Distributed Access Control Policies with Tunable Expressiveness. In *5th IEEE International Workshop on Policies for Distributed Systems and Networks* (2004).

[18] BEERI, C., AND RAMAKRISHNAN, R. On the Power of Magic. In *ACM Symposium on Principles of Database Systems* (1987).

[19] BELARAMANI, N., DAHLIN, M., GAO, L., NAYATE, A., VENKATARAMANI, A., YALAGANDULA, P., AND ZHENG, J. Practi replication. In *USENIX Symposium on Networked Systems Design and Implementation* (2006).

[20] BENJAMIN SZEKELY AND ELIAS TORRES. A Paxon Evaluation of P2. http://www.klinewoods.com/papers/p2paxos.pdf.

[21] BERNSTEIN, P. A., DAYAL, U., DEWITT, D. J., GAWLICK, D., GRAY, J., JARKE, M., LINDSAY, B. G., LOCKEMANN, P. C., MAIER, D., NEUHOLD, E. J., REUTER, A., ROWE, L. A., SCHEK, H.-J., SCHMIDT, J. W., SCHREFL, M., AND STONEBRAKER, M. Future Directions in DBMS Research. *SIGMOD Record 18*, 1 (1989), 17–26.

[22] BERSHAD, B., SAVAGE, S., PARDYAK, P., SIRER, E. G., BECKER, D., FIUCZYNSKI, M., CHAMBERS, C., AND EGGERS, S. Extensibility, Safety and Performance in the SPIN Operating System. In *ACM Symposium on Operating Systems Principles* (1995).

[23] BLAZE, M., FEIGENBAUM, J., AND KEROMYTIS, A. D. The role of trust management in distributed systems security. In *Secure Internet Programming* (1999), pp. 185–210.

[24] CACACE, F., CERI, S., AND HOUTSMA, M. A. W. A survey of parallel execution strategies for transitive closure and logic programs. *Distributed and Parallel Databases 1*, 4 (1993), 337–382.

[25] CALVANESE, D., GIACOMO, G. D., AND VARDI, M. Y. Decidable Containment of Recursive Queries. In *ICDT* (2003).

[26] CASTRO, M., DRUSHEL, P., GANESH, A., ROWSTRON, A., AND WALLACH, D. Secure Routing for Structured Peer-to-peer Overlay Networks. In *Proceedings of Usenix Symposium on Operating Systems Design and Implementation* (2002).

[27] CHANDRASEKARAN, S., COOPER, O., DESHPANDE, A., FRANKLIN, M. J., HELLERSTEIN, J. M., HONG, W., KRISHNAMURTHY, S., MADDEN, S., RAMAN, V., REISS, F., AND SHAH, M. A. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR* (2003).

[28] CHAWATHE, S., GARCIA-MOLINA, H., HAMMER, J., IRELAND, K., PAPAKONSTANTINOU, Y., ULLMAN, J. D., AND WIDOM, J. The TSIMMIS Project: Integration of heterogeneous information sources. In *16th Meeting of the Information Processing Society of Japan* (Tokyo, Japan, 1994).

[29] CHU, D., TAVAKOLI, A., POPA, L., AND HELLERSTEIN, J. M. Entirely Declarative Sensor Network Systems. In *Proceedings of VLDB Conference Demonstrations* (September 2006).

[30] CHU, Y.-H., RAO, S. G., AND ZHANG, H. A Case for End System Multicast. In *Proc. of ACM SIGMETRICS* (2000), pp. 1–12.

[31] CLARK, D. D. The design philosophy of the DARPA internet protocols. In *Proceedings of ACM SIGCOMM Conference on Data Communication* (Stanford, CA, Aug. 1988), ACM, pp. 106–114.

[32] CLOCKSIN, W. F., AND MELISH, C. S. *Programming in Prolog*. Springer-Verlag, 1987.

[33] CODD, E. F. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM 13*, 6 (1970), 377–387.

[34] DABEK, F., LI, J., SIT, E., KAASHOEK, F., MORRIS, R., AND BLAKE, C. Designing a DHT for low latency and high throughput. In *USENIX Symposium on Networked Systems Design and Implementation* (Month 2004).

[35] DEAN, J., AND GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. In *Proceedings of Usenix Symposium on Operating Systems Design and Implementation* (December 2004).

[36] DEERING, S., AND CHERITON, D. R. Multicast routing in datagram internetworks and extended LANs. *ACM Transactions on Computer Systems 8*, 2 (May 1990), 85–111.

[37] DETREVILLE, J. Binder: A logic-based security language. In *IEEE Symposium on Security and Privacy* (2002).

[38] DEWITT, D. J., GERBER, R. H., GRAEFE, G., HEYTENS, M. L., KUMAR, K. B., AND MURALIKRISHNA, M. Gamma - a high performance dataflow database machine. In *Proceedings of VLDB Conference* (1986), pp. 228–237.

[39] DEWITT, D. J., AND GRAY, J. Parallel Database Systems: The Future of High Performance Database Systems. *CACM 35*, 6 (1992), 85–98.

[40] EMULAB. Network Emulation Testbed. http://www.emulab.net.

[41] ENGLER, D., AND MUSUVATHI, M. Model-checking Large Network Protocol Implementations. In *USENIX Symposium on Networked Systems Design and Implementation* (2004).

[42] FEAMSTER, N., AND BALAKRISHNAN, H. Towards a Logic for Wide-Area Internet Routing. In *Proceedings of FDNA-03* (2003).

[43] FEAMSTER, N., AND BALAKRISHNAN, H. Correctness properties for Internet routing. In *Allerton Conference on Communication, Control, and Computing* (Sept. 2005).

[44] FEAMSTER, N., BALAKRISHNAN, H., REXFORD, J., SHAIKH, A., AND VAN DER MERWE, J. The Case for Separating Routing From Routers. In *FDNA* (2004).

[45] FECKO, M., UYAR, M., AMER, P., SETHI, A., DZIK, T., MENELL, R., AND MCMAHON, M. A success story of formal description techniques: Estelle specification and test generation for MIL-STD 188-220. *Computer Communications (Special Edition on FDTs in Practice) 23* (2000).

[46] FIND. NSF NETS Future INternet Design Program. http://find.isi.edu/.

[47] FURFARO, F., GRECO, S., GANGULY, S., AND ZANIOLO, C. Pushing Extrema Aggregates to Optimize Logic Queries. *Inf.Sys. 27*, 5 (2002), 321–343.

[48] GENI. Global Environment for Network Innovations. http://www.geni.net/.

[49] GIBBONS, P. B., KARP, B., KE, Y., NATH, S., AND SESHAN, S. IrisNet: An Architecture for a World-Wide Sensor Web. *IEEE Pervasive Computing 2*, 4 (October-December 2003).

[50] GNUTELLA. http://www.gnutella.com.

[51] GRAEFE, G. Encapsulation of Parallelism in the Volcano Query Processing System. In *Proceedings of ACM SIGMOD International Conference on Management of Data* (1990).

[52] GRIFFIN, T. G., AND SOBRINHO, J. L. Metarouting. In *ACM SIGCOMM* (2005).

[53] GT-ITM. Modelling topology of large networks. http://www.cc.gatech.edu/projects/gtitm/.

[54] GUPTA, A., MUMICK, I. S., AND SUBRAHMANIAN, V. S. Maintaining Views Incrementally. In *Proceedings of ACM SIGMOD International Conference on Management of Data* (1993).

[55] HAAS, Z. J. A New Routing Protocol for the Reconfigurable Wireless Networks. In *IEEE Int. Conf. on Universal Personal Communications* (1997).

[56] HANDLEY, M., GHOSH, A., RADOSLAVOV, P., HODSON, O., AND KOHLER, E. Designing Extensible IP Router Software. In *USENIX Symposium on Networked Systems Design and Implementation* (May 2005).

[57] HANDLEY, M., GHOSH, A., RADOSLAVOV, P., HODSON, O., AND KOHLER, E. Designing IP Router Software. In *USENIX Symposium on Networked Systems Design and Implementation* (2005).

[58] HICKS, M. W., KAKKAR, P., MOORE, J. T., GUNTER, C. A., AND NETTLES, S. PLAN: A packet language for active networks. In *International Conference on Functional Programming* (1998), pp. 86–93.

[59] HUEBSCH, R., CHUN, B., HELLERSTEIN, J., LOO, B. T., MANIATIS, P., ROSCOE, T., SHENKER, S., STOICA, I., AND YUMEREFENDI, A. R. The Architecture of PIER: an Internet-Scale Query Processor. In *CIDR* (2005).

[60] HUEBSCH, R., HELLERSTEIN, J. M., LANHAM, N., LOO, B. T., SHENKER, S., AND STOICA, I. Querying the Internet with PIER. In *Proceedings of VLDB Conference* (Sep 2003).

[61] JEFFERY ULLMAN. Assigning an Appropriate Meaning to Database Logic with Negation. *Computers as Our Better Partners* (1994), 216–225.

[62] JIM, T. SD3: A Trust Management System With Certified Evaluation. In *IEEE Symposium on Security and Privacy* (May 2001).

[63] JIM, T., AND SUCIU, D. Dynamically Distributed Query Evaluation. In *ACM Symposium on Principles of Database Systems* (2001).

[64] JOHNSON, D. B., AND MALTZ, D. A. Dynamic Source Routing in Ad Hoc Wireless Networks. In *Mobile Computing*, vol. 353. 1996.

[65] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The Click Modular Router. 263–297.

[66] KRISHNA P. GUMMADI AND RAMAKRISHNA GUMMADI AND STEVEN D. GRIBBLE AND SYLVIA RATNASAMY AND SCOTT SHENKER AND ION STOICA. The Impact of DHT Routing Geometry on Resilience and Proximity. In *Proceedings of ACM SIGCOMM Conference on Data Communication* (2003).

[67] KRISHNAMURTHY, R., RAMAKRISHNAN, R., AND SHMUELI, O. A Framework for Testing Safety and Effective Computability. *J. Comp. Sys. Sci. 52(1):100-124* (1996).

[68] LAKSHMAN, T. V., NANDAGOPAL, T., RAMJEE, R., SABNANI, K., AND WOO, T. The SoftRouter Architecture. In *HotNets-III* (2004).

[69] LAKSHMINARAYANAN, K., STOICA, I., AND SHENKER, S. Routing as a Service. Tech. Rep. UCB-CS-04-1327, UC Berkeley, 2004.

[70] LAMPORT, L. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems 16*, 3 (May 1994), 872–923.

[71] LAURENT VIEILLE. Recursive Axioms in Deductive Database: The Query-Subquery Approach. In *1st International Conference on Expert Database Systems* (1986).

[72] LEIGHTON, F. T. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, San Mateo, CA, 1992.

[73] LI, J., LOO, B. T., HELLERSTEIN, J., KAASHOEK, F., KARGER, D., AND MORRIS, R. On the Feasibility of Peer-to-Peer Web Indexing and Search. In *International Workshop on Peer-to-Peer Systems* (2003).

[74] LI, N., GROSOF, B. N., AND FEIGENBAUM, J. Delegation Logic: A logic-based approach to distributed authorization. *ACM TISSEC* (Feb. 2003).

[75] LOO, B. T., CONDIE, T., GAROFALAKIS, M., GAY, D. E., HELLERSTEIN, J. M., MANIATIS, P., RAMAKRISHNAN, R., ROSCOE, T., AND STOICA, I. Declarative Networking: Language, Execution and Optimization. In *ACM SIGMOD* (June 2006).

[76] LOO, B. T., CONDIE, T., HELLERSTEIN, J. M., MANIATIS, P., ROSCOE, T., AND STOICA, I. Implementing Declarative Overlays. In *ACM Symposium on Operating Systems Principles* (2005).

[77] LOO, B. T., HELLERSTEIN, J. M., HUEBSCH, R., ROSCOE, T., AND STOICA, I. Analyzing P2P Overlays with Recursive Queries. Tech. Rep. UCB-CS-04-1301, UC Berkeley, 2004.

[78] LOO, B. T., HELLERSTEIN, J. M., HUEBSCH, R., SHENKER, S., AND STOICA, I. Enhancing P2P File-Sharing with an Internet-Scale Query Processor. In *Proceedings of VLDB Conference* (September 2004).

[79] LOO, B. T., HELLERSTEIN, J. M., AND STOICA, I. Customizable Routing with Declarative Queries. In *ACM SIGCOMM Hot Topics in Networks* (2004).

[80] LOO, B. T., HELLERSTEIN, J. M., STOICA, I., AND RAMAKRISHNAN, R. Declarative Routing: Extensible Routing with Declarative Queries. In *Proceedings of ACM SIGMOD International Conference on Management of Data* (2005).

[81] LOO, B. T., HUEBSCH, R., STOICA, I., AND HELLERSTEIN, J. M. The Case for a Hybrid P2P Search Infrastructure. In *International Workshop on Peer-to-Peer Systems* (San Diego, CA, February 2004).

[82] LOO, B. T., KRISHNAMURTHY, S., AND COOPER, O. Distributed Web Crawling over DHTs. Tech. Rep. UCB-CS-04-1305, UC Berkeley, 2004.

[83] MANKU, G., BAWA, M., AND RAGHAVAN, P. Symphony: Distributed hashing in a small world. In *Proc. USITS* (2003).

[84] MILTCHEV, S., PREVELAKIS, V., IOANNIDIS, S., IOANNIDIS, J., KEROMYTIS, A. D., AND SMITH, J. M. Secure and flexible global file sharing. In *USENIX Technical Conference* (June 2003).

[85] MIT. The Chord/DHash Project. http://pdos.csail.mit.edu/chord/.

[86] MORITZ Y. BECKER AND CEDRIC FOURNET AND ANDREW D. GORDON. SecPAL: Design and Semantics of a Decentralized Authorization Language. Tech. Rep. MSR-TR-2006-120, Microsoft Research, 2006.

[87] MOSBERGER, D., AND PETERSON, L. L. Making paths explicit in the Scout operating system. In *Proceedings of Usenix Symposium on Operating Systems Design and Implementation* (1996), ACM Press, pp. 153–167.

[88] O. PAPAEMMANOUIL AND Y. AHMAD AND U. CETINTEMEL AND J. JANNOTTI AND Y. YILDIRIM. Extensible Optimization in Overlay Dissemination Tree. In *Proceedings of ACM SIGMOD International Conference on Management of Data* (1996).

[89] OZSU, M. T., AND VALDURIEZ, P. *Principles of Distributed Database Systems (2nd Edition)*. Prentice Hall, 1999.

[90] OZSU, M. T., AND VALDURIEZ, P. *Principles of Distributed Database Systems, Second Edition*. Prentice Hall, 1999.

[91] PAPADOPOULOS, G. M., AND CULLER, D. E. Monsoon: An explicit token store architecture. In *Proc. ISCA* (May 1990).

[92] PETERSON, L., AND DAVIE, B. *Computer Networks: A Systems Approach*. Morgan-KaufMann, 2003.

[93] PETERSON, L., SHENKER, S., AND TURNER, J. Overcoming the Internet Impasse Through Virtualization. In *HotNets-III* (2004).

[94] PLANETLAB. Global testbed. http://www.planet-lab.org/.

[95] PYTHON PROGRAMMING LANGUAGE. http://www.python.org.

[96] RAGHU RAMAKRISHNAN AND S. SUDARSHAN. Bottom-Up vs Top-Down Revisited. In *Proceedings of the International Logic Programming Symposium* (1999).

[97] RAMAKRISHNAN, R., ROSS, K. A., SRIVASTAVA, D., AND SUDARSHAN, S. Efficient Incremental Evaluation of Queries with Aggregation. In *Proceedings of ACM SIGMOD International Conference on Management of Data* (1992).

[98] RAMAKRISHNAN, R., AND ULLMAN, J. D. A Survey of Research on Deductive Database Systems. *Journal of Logic Programming 23*, 2 (1993), 125–149.

[99] RAMAN, S., AND MCCANNE, S. A model, analysis, and protocol framework for soft state-based communication. In *Proceedings of ACM SIGCOMM Conference on Data Communication* (1999), pp. 15–25.

[100] RAMASUBRAMANIAN, V. Cost-Aware Resource Management for Decentralized Internet Services. Tech. rep., Cornell University, 2007.

[101] RAMASUBRAMANIAN, V., HAAS, Z. J., AND SIRER, E. G. SHARP: A Hybrid Adaptive Routing Protocol for Mobile Ad Hoc Networks. In *ACM MobiHoc* (2003).

[102] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. A Scalable Content Addressable Network. In *Proc. of the ACM SIGCOM Conference* (Berkeley, CA, August 2001).

[103] RHEA, S., GEELS, D., ROSCOE, T., AND KUBIATOWICZ, J. Handling Churn in a DHT. In *USENIX Technical Conference* (June 2004).

[104] ROBBERT, V. R., BIRMAN, K. P., DUMITRIU, D., AND VOGEL, W. Scalable management and data mining using astrolabe. In *International Workshop on Peer-to-Peer Systems* (Cambridge, MA, Mar. 2002).

[105] RODRIGUEZ, A., KILLIAN, C., BHAT, S., KOSTIC, D., AND VAHDAT, A. MACEDON: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks",. In *USENIX Symposium on Networked Systems Design and Implementation* (March 2004).

[106] ROHMER, J., LESCOEUR, R., AND KERISIT, J. M. Alexander Method - A Technique for the Processing of Recursive Axioms in Deductive Databases. *New Generation Computing 4:522-528* (1986).

[107] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. *Lecture Notes in Computer Science 2218* (2001).

[108] R.PALMER, C., GIBBONS, P. B., AND FALOUTSOS, C. ANF: A Fast and Scalable Tool for Data Mining in Massive Graphs. In *ACM SIGKDD* (2002), pp. 102–111.

[109] SAFETYNET. The SafetyNet Project. http://www.cogs.susx.ac.uk/projects/safetynet/.

[110] SESHADRI, P., HELLERSTEIN, J. M., PIRAHESH, H., LEUNG, T. C., RAMAKRISHNAN, R., SRIVASTAVA, D., STUCKEY, P. J., AND SUDARSHAN, S. Cost-based Optimization for Magic: Algebra and Implementation. In *Proceedings of ACM SIGMOD International Conference on Management of Data* (1996).

[111] SHAO, J., BELL, D. A., AND HULL, M. E. C. An Experimental Performance Study of a pipelined recursive query processing strategy. In *International Symposium on Databases for Parallel and Distributed Systems* (1990).

[112] SINGH, A., MANIATIS, P., ROSCOE, T., AND DRUSCHEL, P. Distributed Monitoring and Forensics in Overlay Networks. In *Eurosys* (2006).

[113] SKYPE. Skype P2P Telephony. http://www.skype.com.

[114] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of ACM SIGCOMM Conference on Data Communication* (2001).

[115] STONEBRAKER, M. Inclusion of New Types in Relational Data Base Systems. In *ICDE* (1986).

[116] STONEBRAKER, M., AOKI, P. M., LITWIN, W., PFEFFER, A., SAH, A., SIDELL, J., STAELIN, C., AND YU, A. Mariposa: A wide-area distributed database system. *VLDB Journal 5*, 1 (1996), 48–63.

[117] STONEBRAKER, M., AND HELLERSTEIN, J. M., Eds. *Readings in Database Systems, Third Edition*. Morgan Kaufmann, San Francisco, 1998.

[118] SUDARSHAN, S., AND RAMAKRISHNAN, R. Aggregation and Relevance in Deductive Databases. In *Proceedings of VLDB Conference* (1991).

[119] TENNENHOUSE, D., SMITH, J., SINCOSKIE, W., WETHERALL, D., AND MINDEN, G. A Survey of Active Network Research. In *IEEE Communications Magazine* (1997).

[120] TERRY, D. B., THEIMER, M. M., PETERSEN, K., DEMERS, A. J., SPREITZER, M. J., AND HAUSER, C. H. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *ACM Symposium on Operating Systems Principles* (1995).

[121] THIBAULT, S., CONSEL, C., AND MULLER, G. Safe and Efficient Active Network Programming. In *17th IEEE Symposium on Reliable Distributed Systems* (1998).

[122] TURNER, K. J., Ed. *Using formal description techniques – An Introduction to Estelle, LOTOS and SDL*. Wiley, 1993.

[123] TYSON CONDIE AND JOSEPH M. HELLERSTEIN AND PETROS MANIATIS AND SEAN RHEA AND TIMOTHY ROSCOE. Finally, a Use for Componentized Transport Protocols. In *ACM SIGCOMM Hot Topics in Networks* (2005).

[124] TYSON CONDIE AND VARUN KACHOLIA AND SRIRAM SANKARARAMAN AND JOSEPH M. HELLERSTEIN AND PETROS MANIATIS. Induced Churn as Shelter from Routing-Table Poisoning. In *Network and Distributed System Security* (2006).

[125] VAN DEURSEN, A., KLINT, P., AND VISSER, J. Domain-Specific Languages: An Annotated Bibliography. *SIGPLAN Notices 35*, 6 (2000).

[126] VEEN, A. H. Dataflow machine architecture. *ACM Computing Surveys 18*, 4 (Dec. 1986).

[127] WHALEY, J., AND LAM, M. S. Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *PLDI* (2004).

[128] WILSCHUT, A. N., AND APERS, P. M. G. Pipelining in Query Execution. In *International Conf on Databases, Parallel Architectures and their Applications* (1991).

[129] XINMING OU AND SUDHAKAR GOVINDAVAJHALA AND ANDREW W. APPEL. MulVAL: A Logic-based Network Security Analyzer. In *USENIX Security Symposium* (2006).

[130] YANG, X. NIRA: A New Internet Routing Architecture. In *Proceedings of the ACM SIGCOMM workshop on Future directions in network architecture* (2003).

[131] ZHAO, B. Y., KUBIATOWICZ, J. D., AND JOSEPH, A. D. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Tech. Rep. UCB/CSD-01-1141, UC Berkeley, Apr. 2001.

Appendix A

Proofs

A.1 Proofs for Pipelined Semi-naïve

Symbol	**Representation**
t	A tuple generated at any iteration.
t^i	A tuple generated at the i^{th} iteration.
p_k	The table corresponding to the k^{th} recursive predicate in the rule body.
b_k	A table for the k^{th} base predicate in the rule body.
$FP_S(p)$	Result set for p using SN evaluation.
$FP_P(p)$	Result set for p using PSN evaluation.
$FP_S^i(p)$	Result set for p using SN evaluation at the i^{th} iteration or less.
$FP_P^i(p)$	Result set for p using PSN evaluation for all p tuples that are marked with iteration number i or less.

Table A.1: Proof Notation

In our proofs, we use the notation in Table A.1. Consider a rule with n recursive predi-

cates p_1, p_2,..., p_n and m base predicates $b_1, b_2, ..., b_m$:
$p : -p_1, p_2, ..., p_n, b_1, b_2, ..., b_m$.

For the purposes of the proof of Theorem A.1, we assume that there is a unique derivation for each tuple t.

Claim A.1 $\forall t^i \in FP_S^i(p), \exists t_j \in FP_S^{i-1}(p_j)$ *s.t.* $t : -t_1, t_2, ..., t_n, b_1, b_2, ..., b_m \wedge t \notin FP_S^{i-1}(p)$. *Same for* FP_P.

Theorem A.1 $FP_S(p) = FP_P(p)$

Proof: (By induction). The base case $FP_S^0(p) = FP_P^0(p)$ is trivial since this is the initial set of input p_0 tuples. Assume inductively $FP_S^{i-1}(p) = FP_P^{i-1}(p)$ is true, we show that $FP_S^i(p) = FP_P^i(p)$ using the following two lemmas below.

Lemma A.1 $FP_S^i(p) \subseteq FP_P^i(p)$

Proof: Consider tuple $t^i \in FP_S^i(p)$ derived using SN evaluation $t : -t_1, t_2, ..., t_n, b_1, b_2, ..., b_m$. By Claim A.1, $t_j \in FP_S^{i-1}(p_j) \wedge t \notin FP_S^{i-1}(p)$. One of the input t_j's (t_k) must be in $\triangle \mathrm{p}_k^{old}$ in the SN algorithm. $t_k^{i-1} \in FP_S^{i-1} \Rightarrow t_k^{i-1} \in FP_P^{i-1}$. By the PSN algorithm, t_j^{i-1} must have been enqueued, hence generating t^i. So $t^i \in FP_S^i$.

Lemma A.2 $FP_P^i(p) \subseteq FP_S^i(p)$

Proof: Consider a tuple $t^i \in FP_S^i(p)$ derived using modified PSN evaluation $t : -t_1, t_2, ..., t_n, b_1, b_2, ..., b_m$. From claim A.1, $t_k \in FP_P^{i-1}(p_k) \wedge t \notin FP_P^{i-1}(p)$. By the PSN algorithm, one of t_j's (t_k) is $\triangle t_k^{old,i-1}$. This means that $t_k^{i-1} \in FP_S^{i-1}(p_k) \Rightarrow t_k^{i-1} \in \triangle p_k^{old}$ in the i^{th} iteration of the SN algorithm. This will result in the rule being used to generate t in the i^{th} iteration. Hence, $t^i \in FP_S^i$.

If there are multiple derivations for the same tuple, we can apply the same proof above for Theorem A.1 using the following modified PSN: if there are two derivations t^i and t^j ($j > i$) for the same tuple, the modified PSN algorithm guarantees that t^i is generated by enqueuing t^i even if t^j was previously generated. Note that the modified PSN algorithm leads to repeated inferences, but generates the same results as PSN.

Theorem A.2 *There are no repeated inferences in computing $FP_P(p)$.*

Proof: For linear rules, the theorem is trivially true since we only add a new derived tuple into the PSN queue if it does not exist previously. This guarantees that each invocation of the rule is unique

For non-linear rules, we continue from Theorem A.1's proof. Let $ts(t)$ be the sequence number or timestamp of derived tuple t. Following the proof for Lemma A.1, only the k^{th} rule, where $ts(t_k^{i-1}) = max(ts(t_1^{i-1}), ts(t_2^{i-1}), ..., ts(t_n^{i-1}))$ will be used to generate t_0^i at the inductive step, ensuring no repeated inferences.

A.2 Proofs for Bursty Updates

Let E be the set of all extensional tuples that appear during the execution of a program. Let D be the set of all tuples that can be derived from E (we assume $E \subseteq D$ for simplicity). A tuple $t \in D$ derived by the rule t:-$t_1, t_2, ..., t_n$ has a corresponding *tree fragment*, with parent t and children t_j. The *derivation tree* for D is built by assembling the tree fragments for all possible derivations of tuples in D. We distinguish the multiple tree fragments for multiple derivations of t, but to simplify notation, we use $t, t_1, \ldots$ to name tree nodes. Leaves of this tree are elements of E.

A series of insertions and deletions to the extensional relations is modeled as a sequence of values $t(0), t(1), \ldots, t(j)$ for each $t \in E$, where 1 means present and 0 means absent.

Similarly, for all tree nodes t, we remember the sequence of values (presence or absence) assigned to t by the PSN algorithm after each child change. We write $t(\infty)$ to represent the value of t once the network has quiesced.

Let t be a tree node whose children are $t_1, t_2, ..., t_n$.

Claim A.2 *Along any tree edge $t_k \rightarrow t$, value changes are applied in the order in which t_k's change. This property is guaranteed by PSN's FIFO queue.*

Lemma A.3 *$t(\infty)$ is derived using $t_1(\infty), t_2(\infty), \ldots, t_n(\infty)$.*

Proof: (By induction) $t(0)$ is computed from the initial values of its children. Assume inductively that $t(j-1)$ is derived based on the $(j-1)^{th}$ change in its children. If child t_k changes, $t(j)$ is rederived, and based on Claim A.2, reflects the latest value of t_k. Hence, $t(\infty)$ is derived from the last value of all its children.

Let FP_p be the set of tuples derived using PSN under the bursty model, and FFP_p be the set of tuples that would be computed by PSN if starting from the quiesced state.

Theorem A.3 *$FP_p = FFP_p$ in a centralized setting.*

Proof: We write $t(\omega)$ for the values derived by PSN when its starting state is $e(\infty)$ for $e \in E$. If $\forall t \in D$'s derivation tree, $t(\omega) = t(\infty)$ then $FP_p = FFP_p$. We prove this by induction on the height of tuples in the derivation tree. We define D_i to be all nodes of D's derivation tree at height i, with $D_0 = E$.

In the base case, $\forall t \in D_0$, $t(\infty) = t(\omega)$ by definition of the base tuple values. In the inductive step, we assume that $\forall j < i$, $\forall t \in D_j$, $t(\infty) = t(\omega)$. Consider $t \in D_i$. Based on Lemma A.3, $t(\infty)$ will be derived from the $t_k(\infty)$ values of its children, which by induction are equal to $t_k(\omega)$. Hence $t(\infty) = t(\omega)$.

Claim A.3 *As long as all network links obey FIFO for transmitted messages, Claim A.2 is true for any children of t that are generated using link-restricted Datalog rules.*

Theorem A.4 *$FP_p = FFP_p$ in a distributed setting.*

Proof: With Claim A.3, the proof is similar to that of Theorem A.3.

A.3 Proofs for Soft-State Rules

Let H be the set of hard-state tuples, and S be the set of soft-state tuples. A series of insertions and refreshes to the relations is modeled as a sequence of $t(0), t(1), \ldots, t(j)$ for each $t \in (H \cup S)$, where 1 means present and 0 means absent. We write $t(\infty)$ to represent the value of t when the network has quiesced.

We define the *eventual steady state* of the network after it has quiesced, which includes all hard state tuples $th \in H$ s.t. $th(\infty) = 1$, and all soft-state tuples $ts \in S$ s.t. $ts(\infty) = 1$. Since soft-state tuples have finite lifetimes, in the eventual steady state, all soft-state tuples that satisfy $ts(\infty) = 1$ are periodically refreshed and recomputed at time intervals less than their lifetime. Let $l(ts)$, $p(ts)$ and $r(ts)$ be the lifetime, derivation time and refresh interval of a soft-state tuple $ts \in S$. We observe that $ts(\infty) = 1$ is true iff $l(ts) \geq p(ts) + r(ts)$ is true.

In the rest of the section, we consider the three types of soft-state rules defined in Section 2.5: *pure soft-state rules*, *derived soft-state rules* and *archival soft-state rules*.

A.3.1 Pure Soft-State Rule

A pure soft-state rule is of the form: $s :- s_1, s_2, ...s_m, h_1, h_2, ...h_n$ where there are m soft-state predicates and n hard-state predicates. The rule derives a soft-state predicate s. Let $ts \in s$ be a soft-state tuple derived from $ts :- ts_1, ts_2, ..., ts_m$, $th_1, th_2, ..., th_n$, where $\forall i\ (ts_i \in s_i)$ and $\forall j\ (th_j \in h_j)$. For eventual consistency, we want to show that $ts(\infty)$ is derived using $ts_1(\infty), ts_2(\infty), \ldots, ts_n(\infty)$, $th_1(\infty), th_2(\infty), \ldots, th_n(\infty)$.

The inductive proof is similar to that of Theorem A.3, where we model refreshes instead of deletions of tuples. In addition, we consider the following two cases:

1. **Case 1:** $l(ts) \geq \mathbf{max(}l(ts_1), l(ts_2), ..., l(ts_m)\mathbf{)}$**.** In the first case, the derived soft-state tuple ts has a lifetime that exceeds all of its soft-state inputs $ts_1, ts_2, ..., ts_m$. Given that ts has a finite lifetime, in the eventual steady state of the network, each ts s.t. $ts(\infty) = 1$ is periodically refreshed. At each refresh of one of the input ts_i, ts is re-computed based on the most recent values of $ts_1, ts_2, ...ts_m$, $th_1, th_2, ...th_n$ of the quiesced network. Hence, $ts(\infty)$ is derived using $ts_1(\infty), \ldots, ts_n(\infty)$, $th_1(\infty), \ldots, th_n(\infty)$. In addition, for a given input ts_i, if $ts_i(\infty) = 1$, then $l(ts_i) \geq p(ts_i) + r(ts_i)$ is true. Hence, from the case 1 condition, $l(ts) \geq \max(r(ts_1), l(rs_2), ..., r(ts_m))$. For a given ts that is present at the eventual steady state, $ts(\infty) = 1$ is true until the next refresh of one of the input ts_i.

2. **Case 2:** $l(ts) < \mathbf{max(}l(ts_1), l(ts_2), ..., l(ts_m)\mathbf{)}$**.** Similarly, in the eventual steady state of the network, at each refresh of ts_i, ts will be derived based on the most recent values of $ts_1, ts_2, ...ts_m$, $th_1, th_2, ...th_n$ of the quiesced network. However, from the case 2 condition, it is possible that $l(ts) < \max(r(ts_1), r(ts_2), ..., r(ts_m))$. This means that ts may expire before the next refresh of any input ts_i. Hence, $ts(\infty)$ will oscillate between 0 and 1 in the eventual steady state of the network.

In conclusion, we can achieve eventual consistency for pure soft-state rules. However, in case 2, the eventual state of the network may not be stable and some soft-state derivations will oscillate being derived and timing out. To avoid such oscillations, we can disallow rules that satisfy the second condition. This can be enforced via syntactic checks to ensure the lifetime of the derived soft-state head exceeds the lifetime of all the soft-state body predicates.

A.3.2 Derived Soft-State Rule

We next consider a derived soft-state rule of the form $s:-h_1, h_2, ...h_n$, which takes as input n hard-state predicates, and derive a soft-state predicate s. Give that the rule body consist only of hard-state predicates, the proof is similar to that of Theorem A.3. However, given the lack of refreshes in the inputs and the fact that all derived $ts \in s$ tuples have finite lifetimes, $ts(\infty) = 0$ in the eventual steady state.

A.3.3 Archival Soft-state Rules

Last, we consider archival soft-state rule of the form $h:-s_1, s_2, ...s_m$, $h_1, h_2, ...h_n$, where there are m soft-state predicates and n hard-state predicates. The rule derives a hard-state predicate h. Consider a hard-state tuple $th \in H$ derived using $th:-ts_1, ts_2, ...ts_m$, $th_1, th_2, ...th_n$. Unlike soft-state tuples, th needs to be explicitly deleted. Given that there are no cascaded deletions in soft-state rules, th is not deleted even when one of $ts_1, ts_2, ..., ts_m$ has expired. Hence, $th(\infty)$ is not derived using $ts_1(\infty), \ldots, ts_n(\infty)$, $th_1(\infty), \ldots, th_n(\infty)$.

Appendix B

Examples of Declarative Overlays

B.1 Narada

We provide an executable *NDlog* implementation of Narada's mesh maintenance algorithms that includes (1) membership list maintenance, and (2) liveness checks on neighbors.

```
/* Materialized table declarations */
materialize(sequence, infinity, 1, keys(2)).
materialize(env, infinity, infinity, keys(2,3)).
materialize(neighbor, infinity, infinity, keys(2)).
materialize(member, 120, infinity, keys(2)).

/* Initial facts */
e1 neighbor(@X,Y) :- periodic(@X,E,0,1), env(@X,H,Y), H = "neighbor".
e2 member(@X,A,S,T,L) :- periodic(@X,E,0,1), T = f_now(), S = 0, L = 1, A = X.
e3 member(@X,Y,S,T,L) :- periodic(@X,E,0,1), neighbor(@X,Y), T = f_now(),
                        S = 0, L = 1.
```

```
e4 sequence(@X,Sequence) :- periodic(@X,E,0,1), Sequence = 0.
```

/* Membership list maintenance */

```
r1 refreshEvent(@X) :- periodic(@X,E,5).
r2 refreshSeq@X(X,NewS) :- refreshEvent@X(X), sequence@X(X,S), NewS = S + 1.
r3 sequence@X(X,NewS) :- refreshSeq@X(X,NewS).
r4 refreshMsg(@Y,X,NewS,Addr,AS,ALive) :- refreshSeq(@X,NewS),
                        member(@X,Addr,AS,Time,ALive),
                        neighbor(@X,Y).
r5 membersCount(@X,Addr,AS,ALive,count<*>) :- refreshMsg(@X,Y,YS,Addr,AS,ALive),
                        member(@X,Addr,MyS,MyTime,MyLive),
                        X != Addr.
r6 member(@X,Addr,AS,T,ALive) :- membersCount(@X,Addr,AS,ALive,C),
                        C = 0, T = f_now().
r7 member(@X,Addr,AS,T,ALive) :- membersCount(@X,Addr,AS,ALive,C),
                        member(@X,Addr,MyS,MyT,MyLive),
                        T = f_now(), C > 0, MyS < AS.
r8 neighbor(@X,Y) :- refresh(@X,Y,YS,A,AS,L).
```

/* Liveness checks on neighbors */

```
l1 neighborProbe(@X) :- periodic(@X,E,1).
l2 deadNeighbor(@X,Y) :- neighborProbe(@X), T = f_now(),
                        neighbor(@X,Y), member(@X,Y,YS,YT,L), T - YT > 20.
l3 delete neighbor(@X,Y) :- deadNeighbor(@X,Y).
l4 member(@X,Neighbor,DeadSequence,T,Live) :- deadNeighbor(@X,Neighbor),
                        member(@X,Neighbor,S,T1,L), Live = 0,
```

DeadSequence = S + 1, T = f_now().

B.2 P2-Chord

Here we provide the full *NDlog* specification for Chord. This specification deals with lookups, ring maintenance with a fixed number of successors, finger-table maintenance and opportunistic finger table population, joins, stabilization, and node failure detection.

/* Materialized table declarations */
materialize(nodeID, infinity, 1, keys(1)).
materialize(landmark, infinity, 1, keys(1)).
materialize(finger, 180, 160, keys(2)).
materialize(uniqueFinger, 180, 160, keys(2)).
materialize(bestSucc, 180, 1, keys(1)).
materialize(succ, 30, 16, keys(2)).
materialize(pred, infinity, 1, keys(1)).
materialize(join, 10, 5, keys(1)).
materialize(pendingPing, 10, infinity, keys(3)).
materialize(fFix, 180, 160, keys(2)).
materialize(nextFingerFix, 180, 1, keys(1)).

/* Initial bootstrapping facts */
i1 pred(@NI,P,PI) :- periodic(@NI,E,0,1), P = “NIL”, PI = ”NIL”.
i2 nextFingerFix(@NI, 0) :- periodic(@NI,E,0,1).
i3 node(@NI,N) :- periodic(@NI,E,0,1), env(@NI,H,N), H = ”node”.

```
i4 landmark(@NI,LI) :- periodic(@NI,E,0,1), env(@NI,H,LI), H = "landmark".


/* Lookups */
l1 lookupResults(@R,K,S,SI,E) :- nodeID(@NI,N), lookup(@NI,K,R,E),
                              bestSucc(@NI,S,SI), K in (N,S].
l2 bestLookupDist(@NI,K,R,E,min<D>) :- nodeID(@NI,N),
                              lookup(@NI,K,R,E), finger(@NI,I,B,BI),
                              D = K - B - 1, B in (N,K).
l3 lookup(min<@BI>,K,R,E) :- nodeID(@NI,N),
                              bestLookupDist(@NI,K,R,E,D), finger(@NI,I,B,BI),
                              D = K - B - 1, B in (N,K).


/* Best successor and first finger selection */
n1 newSuccEvent(@NI) :- succ(@NI,S,SI).
n2 newSuccEvent(@NI) :- deleteSucc(@NI,S,SI).
n3 bestSuccDist(@NI,min<D>) :- newSuccEvent(@NI),nodeID(@NI,N), succ(@NI,S,SI),
                              D = S - N - 1.
n4 bestSucc(@NI,S,SI) :- succ(@NI,S,SI), bestSuccDist(@NI,D), nodeID(@NI,N),
                              D = S - N - 1.
n5 finger(@NI,0,S,SI) :- bestSucc(@NI,S,SI).


/* Successor eviction */
s1 succCount(@NI,count<*>) :- newSuccEvent(@NI), succ(@NI,S,SI).
s2 evictSucc(@NI) :- succCount(@NI,C), C > 4.
s3 maxSuccDist(@NI,max<D>) :- succ(@NI,S,SI),
                              nodeID(@NI,N), evictSucc(@NI),
                    D = S - N - 1.
s4 delete succ(@NI,S,SI) :- nodeID(@NI,N), succ(@NI,S,SI),
                    maxSuccDist(@NI,D), D = S - N - 1.


/* Finger fixing */
f1 fFix(@NI,E,I) :- periodic(@NI,E,10), nextFingerFix(@NI,I).
f2 fFixEvent(@NI,E,I) :- fFix(@NI,E,I).
f3 lookup(@NI,K,NI,E) :- fFixEvent(@NI,E,I), nodeID(@NI,N), K = 0x1I << I + N.
f4 eagerFinger(@NI,I,B,BI) :- fFix(@NI,E,I), lookupResults(@NI,K,B,BI,E).
f5 finger(@NI,I,B,BI) :- eagerFinger(@NI,I,B,BI).
f6 eagerFinger(@NI,I,B,BI) :- nodeID(@NI,N),
                    eagerFinger(@NI,I1,B,BI), I = I1 + 1,
                    K = 0x1I << I + N, K in (N,B), BI != NI.
f7 delete fFix(@NI,E,I1) :- eagerFinger(@NI,I,B,BI), fFix(@NI,E,I1),
                    I > 0, I1 = I - 1.
f8 nextFingerFix(@NI,0) :- eagerFinger(@NI,I,B,BI), ((I = 159)  (BI = NI)).
f9 nextFingerFix(@NI,I) :- nodeID(@NI,N),
                    eagerFinger(@NI,I1,B,BI), I = I1 + 1,
                    K = 0x1I << I + N, K in (B,N), NI != BI.
f10 uniqueFinger(@NI,BI) :- finger(@NI,I,B,BI).


/* Join network */
j1 joinEvent(@NI,E) :- periodic(@NI,E,1,2).
j2 join(@NI,E) :- joinEvent(@NI,E).
j3 joinReq@LI(LI,N,NI,E) :- joinEvent(@NI,E), nodeID(@NI,N),
                    landmark(@NI,LI), LI != "NIL".
```

```
j4 succ(@NI,N,NI) :- landmark(@NI,LI), joinEvent(@NI,E), nodeID(@NI,N), LI = “NIL”.
j5 lookup@LI(LI,N,NI,E) :- joinReq@LI(LI,N,NI,E).
j6 succ(@NI,S,SI) :- join(@NI,E), lookupResults(@NI,K,S,SI,E).

/* Stabilization */
sb1 succ(@NI,P,PI) :- periodic(@NI,E,10), nodeID(@NI,N),
                      bestSucc(@NI,S,SI), pred(@SI,P,PI),
                      PI != “NIL”, P in (N,S).
sb2 succ(@NI,S1,SI1) :- periodic(@NI,E,10), succ(@NI,S,SI), succ(@SI,S1,SI1).
sb3 pred@SI(SI,N,NI) :- periodic(@NI,E,10), nodeID(@NI,N),
                      succ(@NI,S,SI), pred(@SI,P,PI),
                      node(@SI,N1), ((PI = “NIL”)  (N in (P,N1))).

/* Ping-Pong messages to neighbors */
pp1 pendingPing(@NI,SI,E1,T) :- periodic(@NI,E,5), succ(@NI,S,SI),
                      E1 = f_rand(), SI != NI, T = f_now().
pp2 pendingPing(@NI,PI,E1,T) :- periodic(@NI,E,5), pred(@NI,P,PI),
                      E1 = f_rand(), PI ! = “NIL”, T = f_now().
pp3 pendingPing(@NI,FI,E1,T) :- periodic(@NI,E,5), uniqueFinger(@NI,FI),
                      E1:=f_rand(), T:=f_now().
pp4 pingResp(@RI,NI,E) :- pingReq(@NI,RI,E).
pp5 pingReq(@PI,NI,E) :- periodic(@NI,E1,3),
                      pendingPing(@NI,PI,E,T).
pp6 delete pendingPing(@NI,SI,E,T) :- pingResp(@NI,SI,E), pendingPing(@NI,SI,E,T).
```

/* Failure Detection */

fd1 nodeFailure(@NI,PI,E1,D) :- periodic(@NI,E,1), pendingPing(@NI,PI,E1,T),

T1 = f_now(), D = T-T1, D $>$ 7.

fd2 delete pendingPing(@NI,PI,E,T) :- nodeFailure(@NI,PI,E,D), pendingPing(@NI,PI,E,T).

fd3 deleteSucc(@NI,S,SI) :- succ(@NI,S,SI), nodeFailure(@NI,SI,E,D).

fd4 delete succ(@NI,S,SI) :- deleteSucc(@NI,S,SI).

fd5 pred(@NI,"NIL","NIL") :- pred(@NI,P,PI), nodeFailure(@NI,PI,E,D).

fd6 delete finger(@NI,I,B,BI) :- finger(@NI,I,B,BI), nodeFailure(@NI,BI,E,D).

fd7 delete uniqueFinger(@NI,FI) :- uniqueFinger(@NI,FI), nodeFailure(@NI,FI,E,D).