Building and Optimizing Declarative Networked Systems

by

David Chiyuan Chu

B.S. (University of Virginia) 2004
M.S. (University of California, Berkeley) 2005

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:

Dr. Joseph M. Hellerstein, Chair
Dr. Ion Stoica
Dr. William E. Dietrich

Spring 2009

The dissertation of David Chiyuan Chu is approved.

Chair Date

Date

Date

University of California, Berkeley

Spring 2009

Building and Optimizing Declarative Networked Systems

by

David Chiyuan Chu

Abstract

Building and Optimizing Declarative Networked Systems

by

David Chiyuan Chu

Doctor of Philosophy in Computer Science

University of California, Berkeley

Dr. Joseph M. Hellerstein, Chair

In the face of progressively diverse networking technologies and application traffic, it is increasingly infeasible to custom engineer networked systems for each scenario. Moreover, an expanding class of networks, networked embedded systems, are very difficult to program, yet require a high degree of per-deployment programming customization.

We investigate a declarative approach to building and optimizing networked systems, with emphasis on networked embedded systems. Our findings indicate that ideas from data management may yield dividends for the design of networked systems in two key areas: (1) declarative interfaces for simplicity yet breadth of expressiveness, and (2) automatic optimizations for automatic performance improvements on the users' behalf.

This dissertation reports on three efforts. First, we designed and implemented DSN: a declarative language, runtime and compiler for networked embedded systems. The new logic-based language in DSN has been highly intuitive for programming – in one case, an algorithm designers' pseudocode mapped nearly line-for-line to working DSN code. Typically, lines-of-code are reduced by an order of magnitude vs. implementations in traditional embedded languages. We built a complementary compiler and runtime that showed negligible performance drop off

vs. hand-tuned C implementations. As a result, we have been able to build whole system stacks – save for device drivers – entirely declaratively in under a hundred lines of code.

Next, we designed and implemented `netopt`, a network optimizer that relieves programmers from having to manually solve two general networking problems: rendezvous and proxy selection. As part of this effort, we created novel program analysis and transformation algorithms to automatically select optimal communication rendezvous and proxies based on traffic and network conditions. When combined with either the DSN system, or similar systems for PC-class devices, user programs get 1-2 orders of magnitude performance improvement without need for programmers' assistance.

Lastly, we designed and implemented `wireless-netopt`, an extension of `netopt` for wireless networking. `wireless-netopt` includes three wireless network optimizations from different layers of the networking stack. We show that the declarative interface readily supports such new domain-specific optimizations. Furthermore, these optimizations can be applied automatically and without added programmer effort, benefiting programs by 2× in energy savings.

Dr. Joseph M. Hellerstein
Dissertation Committee Chair

To my wife, Wynn Nyane.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my advisor, Joe Hellerstein. His driving energy and excellent insights have made our collaboration very enjoyable. Joe has consistently engaged my intellectual curiosity, and fostered my research. I am very glad to have worked closely with him for the duration of my dissertation.

Ion Stoica and William Dietrich graciously extended their time to serve on my dissertation committee. I also owe many thanks to my undergraduate mentor, Marty Humphrey, for investing in me, and exposing me to research at an early stage.

I have had the fortune to work with several excellent research mentors in industry: Wei Hong, Feng Zhao and Jie Liu. They have each broadened my research perspective, and guided me toward exciting, yet practical problems. I would also like to thank collaborators Daniel Malmon and Joel Johnson at the United States Geological Survey, and Phil Levis at Stanford. Our work together on geological process monitoring has enhanced my appreciation for building and deploying embedded systems.

Next, I would like to thank my office mates and members of the database group, among them: Tyson Condie, Daisy Zhe Wang, Alexandra Meliou, Eirinaios Michelakis, Rusty Sears, Shawn Jeffery, Eugene Wu, Amol Deshpande, Ryan Huebsch, Boon Thau Loo, Fred Reiss, Peter Alvaro, Kuang Chen and Neil Conway. I would also like to recognize many good friends at Berkeley, among them: Lucian Popa, Arsalan Tavakoli, Kevin Klues, Prabal Dutta, Xiaofan (Fred) Jiang, Jay Taneja, Jorge Ortiz, Stephen Dawson-Haggerty, Jaein Jeong, Sukun Kim, Jeremy Schiff, Rabin Patra, Jayanth Kannan, Karl Chen, Cheng Tien Ee, Jonathan Hui, Steve Lee, Edmund Wong, Steve Sinha, and Andreas Schmidt. These colleagues embody the rich discourse that has defined my Berkeley experience.

The Berkeley undergraduate researchers with whom I have collaborated have been eminently capable. Among them are: Kaisen Lin, Giang Nguyen, Alex Linares, Michelle Au, Ashik Manandhar and Akshay Kannan. I would also like to recognize the visiting scholars with

whom I have collaborated, among them: Tsung-te (Ted) Lai and Chun-Ying Huang. It's been a pleasure to work with such distinguished individuals.

I would like to thank my parents and family. My parents have always sacrificed for my education without hesitation. As I mature, my appreciation for the nurturing environment I was afforded continues to increase.

I reserve the deepest gratitude for my wife, Wynn Nyane. Three of her qualities have been foundational to my career. Her on point perception has been a consistently accurate field guide to life. Her unyielding support has buttressed my confidence when I have needed it most. Her infectious smile makes me eager to tackle the opportunities ahead of us.

Chapter 1

Introduction

Networks are growing increasingly diverse. The causes of this diversity stem from both novel workload demands and new resource availability. From above, we are witnessing many new applications such as Video-on-Demand, multi-party telephony, and distributed sensing and actuation. From below, we are seeing infrastructure composed from satellites, cellular networks, urban WiFi, and short-range radio like 802.15.4 and bluetooth. As applications and networks evolve, the leading methodology for addressing workload and resource diversities has been to engineer custom networked systems one environment at a time. This approach may be difficult to scale; the combinations of environments and applications that will coexist are poised to outstrip the ability to address each combination individually.

Wireless sensor networking is one class of networking that exemplifies this situation. Over the past five to ten years, wireless sensors have been employed in novel and varied uses – from forecasting nature's course in landslide and animal husbandry prediction (1; 2), to monitoring large man-made structures, such as the modern computer data center and San Francisco's Golden Gate Bridge (3; 4). These small, untethered and embedded devices have and continue to expose new insight about our physical environments by obviating the wired infrastructure that has encumbered traditional distributed monitoring.

However, the short history of sensornets indicates that each new deployment is very taxing

to bring online. Practitioners in the field regard rolling out each new deployment as nontrivial, as evidenced by the ample reporting on deployment experiences in the literature, and by the anecdotal statistics on man-months required per deployment (5; 6; 7; 8).

In large part, deployment challenges stem from the fact that sensor networks are notoriously difficult to program, given that they encompass the complexities of both distributed and embedded systems. Many programming tools and frameworks have attempted to address programmability (9). In this wide spectrum of approaches, some have even eschewed programmability altogether, opting instead for a black-box interface to entire sensor networks (10). Unfortunately, sensornets are embedded, and embedded systems are fundamentally closely coupled with their application and the physical world. As a result, some degree of deployment-specific customization is frequently necessary. Moreover, deployment users often want to retask their systems weeks, months or even several years later (11; 12). Ultimately, programmability is, and will remain, essential to networked systems, and especially sensor networks.

Fortunately, we can look to the field of data management, where analogous challenges existed several decades ago. Current networking design methodology bears some resemblance to early pre-relational data management. In that context, application developers initially encoded data requests (what is wanted) jointly with data access (how to get it). This approach ultimately proved unsustainable for two reasons. First, new applications and their request workloads continually evolved. Second, available resources, specifically memory and disk technology, advanced unabated. Such workload and resource evolution constantly pressured application developers to rethink data access. As a result, the philosophy of data independence emerged to decouple the what from the how, and automated optimization of data access flourished and is now commonplace (13).

This dissertation takes the lessons of data independence and applies them to networking, with an emphasis on sensornets. Fundamentally, we ask whether a declarative approach can help us program networked systems. As a result, two hypothesis emerge.

- Declarative languages are high-level languages. Therefore, declarative user code should be more concise and easier to understand than imperative code. We quantify conciseness by

counting lines of code. While code complexity is a challenging metric to quantify, orders of magnitude differences in program lines-of-code suggest true qualitative differences in code complexity. Furthermore, writing programs declaratively should be easier than writing them imperatively. We qualitatively assess the ease of programming by writing many programs, and by building entire systems declaratively.

- Declarative languages separate user specification from system execution. Therefore, declarative programs should be amenable to automatic optimization. Users should realize automatic performance gains without additional programming effort. We assess the capacity for automatic optimization by comparing the performance of unoptimized declarative code to optimized declarative code.

The bulk of this dissertation is dedicated to testing these hypothesis for networked systems.

1.1 Contributions

We have developed a database-inspired compiler, optimizer and runtime for wireless sensor networks. Diversity is pronounced in sensornets, and resources are rarely over-provisioned, and thus optimized executions are very important. In addition, we also show that our optimizations are applicable to many PC-class networking problems.

Conceptually, the contributions of this dissertation stem from viewing networking data and application data through the same lens. Specifically, we built DSN, a declarative language compiler and runtime for sensornets; built `netopt`, a network optimizer architecture, and designed several new optimization algorithms to solve networking problems; and built `wireless-netopt`, an extension of `netopt` incorporating three commonly used wireless-specific optimizations. These contributions are described in more detail below.

- **DSN language, compiler and runtime.** We developed a new language, `netlog`, as our declarative programming interface (Chapter 2.2). `netlog` is a dialect of the venerable logic language Datalog with adaptations for both networked and embedded computing. We

show that the language is a natural fit not only for end-users, but also for systems builders. We accomplish this by specifying several very different classes of traditional sensor network protocols, services and applications entirely declaratively – these include tree routing, geographic routing, link estimation, data collection, event tracking, version coherency, and localization (Chapter 2.3). To our knowledge, this is the first time these disparate sensornet tasks have been addressed by a single high-level programming environment. In addition, the programs are typically very concise, in one case matching designer pseudocode nearly line for line. Moreover, the declarative approach accommodates the desire for architectural flexibility and simple management of limited resources (Chapter 2.4). These results suggest that declarative languages are well-suited to sensornet programming.

To realize `netlog` programming, we designed and implemented DSN, a declarative sensor network platform consisting of a sensornet compiler and runtime (Chapter 2.5-2.6). Notably, DSN-compiled executables show no runtime performance drop-off compared to C implementations across a variety of applications. The price has been a manageable (typically 3-5KB) increase in memory footprint (Chapter 2.7).

- **netopt optimizer.** We built `netopt`, a network optimizer into the compiler for DSN and for Evita Raced (14), a declarative language compiler and runtime for PC-class devices. When `netopt` optimizations are enabled, declarative programs often increase performance by an order of magnitude, and sometimes by two orders of magnitude over unoptimized programs (Chapter 3.6). These optimizations are automatically applied to source programs – the programmer need not invest added effort.

  `netopt` focuses on rendezvous and proxy selection in the network (Chapter 3.3-3.4). Rendezvous and proxy selection are common design decisions in many networked systems, and optimization algorithms to solve these problems are well known (15). Some examples where rendezvous and proxy selection are important include sensornet event detection, content distribution networks, publish-subscribe systems, client-server state management, Internet QoS, and ad hoc routing (16; 17; 18; 19; 20; 21; 22). Yet all too often, rendezvous and proxy selection decisions are either (1) made with much manual effort on a case-by-case basis, or (2) ignored entirely. In the latter case, developers write naive programs

that do not account for possible changes in rendezvous and proxy selection. We designed and implemented algorithms to automatically (1) identify rendezvous and proxy selection opportunities from naive program source, (2) rewrite naive program source to expose rendezvous and proxy choices, and (3) call out to graph algorithms to search for optimal configurations. The result is that users can write simple programs, and the optimizer takes care of program tuning on the users' behalf.

The `netopt` algorithms are inspired by traditional database query optimization, namely recursive selection push down. Beyond the previously mentioned performance benefits that programmers receive, a conceptual contribution of the work is demonstrating the fruitful cross-pollination of traditional database query optimization and network engineering.

- **wireless-netopt optimizer.** `wireless-netopt` extends the original optimizations in `netopt` with three optimizations that are simple, yet important to the wireless domain: (1) choosing whether or not to redirect traffic to an intermediary, (2) deciding between broadcast or unicast transmission, and (3) setting nodes' channel polling intervals. These optimizations span the networking stack. To date, cross-layer network optimizations are often performed in isolation of one another, in part due to limited visibility between network stack abstraction boundaries. While abstractions remain available to DSN programmers, the declarative nature of the DSN language means that the optimizer retains the ability to perform analysis across network stack boundaries. We evaluated the algorithmic, architectural, and performance implications of such cross-layer optimization for a network optimizer. Our results indicate that the `wireless-netopt` optimizer is capable of supporting isolated as well as cross-layer optimization algorithms, and cross-layer performance improvements exceed isolated optimization by an order of magnitude in energy savings.

1.2 Organization

The remainder of this dissertation is organized as follows. Chapter 2 presents the design and implementation of DSN, the language, compiler and runtime for sensornets. Chapter 3 presents the design of several network optimization algorithms, and the design and implementation of `netopt`, the network optimizer. Chapter 4 presents the design and implementation of `wireless-netopt`, an extension of `netopt` with several wireless network optimizations. Chapter 5 discusses related work. Chapter 6 discusses current limitations of DSN, `netopt` and `wireless-netopt`, and interesting directions for future work. Chapter 7 concludes the dissertation.

Chapter 2

Declarative Sensor Networks

2.1 Motivation

Sensornets and the environments they monitor are fundamentally closely coupled – after all, we are embedding devices in the physical world. As a result, deep customization is often necessary for each sensornet deployment (12; 3; 4; 5; 6; 7; 23). Moreover, even with deployment specific customization, end users often want to reconfigure functionality after initial exploratory anlaysis, or on a periodic basis (11; 12).

However, despite years of research, sensornet programming is still very hard. Most sensornet protocols and applications continue to be written in low-level embedded languages, and must explicitly contend with issues of wireless communication, limited resources, and asynchronous event processing. This kind of low-level programming is challenging even for experienced programmers, and hopelessly complex for typical end users. The design of a general-purpose, easy-to-use, efficient programming model remains a major open challenge in the sensor network community.

This chapter presents the design and implementation of a *declarative sensor network (DSN)* platform: a programming language, compiler and runtime system to support declarative speci-

fication of wireless sensor network applications. Declarative languages are known to encourage programmers to focus on program outcomes (*what* a program should achieve) rather than implementation (*how* the program works). Until recently, however, their practical impact was limited to core data management applications like relational databases and spreadsheets (24). This picture has changed significantly in recent years: declarative approaches have proven useful in a number of new domains, including program analysis (25), trust management (26), and distributed system diagnosis and debugging (27; 28). Of particular interest in the sensornet context is recent work on *declarative networking*, which presents declarative approaches for protocol specification (29) and overlay network implementation (30). In these settings, declarative logic languages have been promoted for their clean and compact specifications, which can lead to code that is significantly easier to specify, adapt, debug, and analyze than traditional procedural code.

Our work on declarative sensor networks originally began with a simple observation: by definition, sensor network programmers must reason about both data management and network design. Since declarative languages have been successfully applied to both these challenges, we expected them to be a good fit for the sensornet context. To evaluate this hypothesis, we developed a declarative language that is appropriate to the sensornet context, and then developed fully-functional declarative specifications of a broad range of sensornet applications. In this work, we present some examples of these declarative specifications: a data collection application akin to TinyDB (31), a software-based link estimator, several multi-hop routing protocols including spanning-tree and geographic routing, the version coherency protocol Trickle (32), the localization scheme NoGeo (33) and an event tracking application faithful to a recently deployed tracking application (34). The results of this exercise provide compelling evidence for our hypothesis: the declarative code naturally and faithfully captures the logic of even sophisticated sensornet protocols. In one case the implementation is almost a *line-by-line translation* of the protocol inventors' pseudocode, directly mapping the high-level reasoning into an executable language.

Establishing the suitability of the declarative approach is of course only half the challenge: to be useful, the high-level specifications have to be compiled into code that runs efficiently on

resource-constrained embedded nodes in a wireless network. We chose to tackle this issue in the context of Berkeley Motes and TinyOS, with their limited set of hardware resources and lean system infrastructure. Our evaluation demonstrates both the feasibility and faithfulness of DSN for a variety of programs, showing that the resulting code can run on resource-constrained nodes, and that the code does indeed perform according to specification in both testbed and simulation environments.

2.1.1 Declarative Sensornets: A Natural Fit?

While declarative languages are famous for hiding details from the programmer, they are correspondingly infamous for preventing control over those details. Our experience confirms this, and suggests DSN is not well-suited to all sensornet tasks. Like most database-style languages, the language in DSN is not adept at natively manipulating opaque data objects such as timeseries, matrices and bitmasks; nor is it fit for providing real-time guarantees. In addition, as a variant of Datalog, the core of DSN's language is limited to expressing the class of programs that are computable in polynomial time. As a result of these shortcomings, we have taken the pragmatic approach and provided flexible mechanisms to interface to external code, as discussed in Section 2.2.

That said, we have been pleasantly surprised at the breadth of tasks that we have been able to program concisely within DSN. In Section 2.4.1 we describe a fully-functioning data collection implementation expressed entirely declaratively, save for natively implemented device drivers. Also, in Section 2.4.2 we discuss features of our language that allow for simple declarative management of resources, a vital concern for sensornets. A goal of our implementation is to allow programmers the flexibility to choose their own ratio of declarative to imperative code, while attempting in our own research to push the boundaries of the declarative language approach as far as is natural.

DSN's declarative approach does not fit exclusively into any one of the existing sensornet programming paradigm clusters. In its family of expressible network protocols, DSN can model spatial processing inherent to region-based group communication (35; 36; 37). While

DSN's execution runtime consists of a chain of database operations resembling the operator-based data processing common to dataflow models (38; 39), DSN users write in a higher-level language. DSN also provides the runtime safeguards inherent to database systems and virtual machines (31; 40; 41). Section 5.1 discusses this work's relationship to other high level sensornet languages in the literature in detail.

This chapter is organized as follows. Sections 2.2 and 2.3 outline the declarative language, and provide examples of a variety of services and applications. Section 2.4 discusses additional features of DSN that suit sensor networks. Sections 2.5 and 2.6 present an architectural overview of the system, along with implementation concerns. Section 2.7 discusses evaluation methodology, measurements and results. Sections 2.8 and 2.9 outlines limitations of our system and summarizes this chapter's results.

2.2 An Introduction to `netlog`

In this section we give an overview of our declarative language `netlog`. `netlog` is a dialect of Datalog, a classic deductive database query language (42). `netlog` supports features fundamental to sensornets: networked execution, and interfacing with the physical world. The typical DSN user, whether an end-user, service implementor or system builder, writes only a short declarative specification using `netlog`.

The main language constructs are *relations*, *tuples*, *facts* and *rules*. `netlog` programs consist of period-terminated statements. The following is a representative, but simplified, example of these elements.

```
% rule
temperatureLog(Time, TemperatureVal) :-
  thermometer(TemperatureVal),
  TemperatureVal > 15,
  timestamp(Time).

% facts
thermometer(24).
timestamp(day1).
```

The *relations* above are temperatureLog, thermometer and timestamp. These are analogous to the tables of a database. *Tuples* are relations with all parameters assigned. A *fact*, such as thermometer(24), instantiates a tuple at the beginning of execution.

A *rule* instantiates tuples based on the truth of a logical expression. Each rule consists of a *head* and *body* that appear on the left and right respectively of the rule's deduction symbol (":–"). The body defines a set of preconditions, which if true, instantiates tuple(s) in the head. Viewed operationally, this model is extremely simple: relations are best thought of as tables with columns in a database, tuples as table rows with values assigned to columns, and rules simply generate new table rows from existing table rows.

For example, temperatureLog(TemperatureVal,Time) is the head of the rule above, while the thermometer and timestamp relations form its body. This rule creates a temperatureLog tuple when there exist thermometer and timestamp tuples and the temperatureVal of the thermometer tuple is greater than 15.

The two facts establish the time and thermometer reading. The tuples given by these facts make the rule body true, so the rule creates a new tuple temperatureLog(24, day1). Following Datalog convention, relations and constants start with lowercase while variables start with upper case letters: TemperatureVal is a variable, while day1 is a constant.

Unification further limits valid head tuples by detecting repeated variables among relation

parameters in the body. For example, in the following program, evidence is true if temperatureLog and pressureLog each have tuples whose first parameters, both named Time, match.

```
evidence(TemperatureVal, PressureVal) :-
  temperatureLog(Time, TemperatureVal),
  pressureLog(Time, PressureVal).

pressureLog(day1,1017).
pressureLog(day2,930).
```

Combined with the listed pressureLog facts and the temperatureLog(day1,24) tuple yielded from the previous example, the rule results in evidence(24,1017). In relational database terminology, unification is an equality join.

2.2.1 Distributed Execution

In a fashion similar to (30), each relation is horizontally partitioned such that each node holds a subset of the tuples of each relation. A relation's first argument is the horizontal partition key, or *location specifier*. This is indicated explicitly by marking each tuple's first argument with the at symbol "@". A single rule may involve tuples hosted at different nodes. For example, the following facts are hosted at the node whose identifier is node1.

```
consume(@node1, base).
produce(@node1, data1).
```

Distributed rules specify relations with different location specifiers.

```
store(@Y, Object) :- produce(@X, Object), consume(@X,Y).
```

With the two facts above, this instantiates store(@base,data1). The different nodes that appear in a rule such as base and node1 have to be within local communication range for the tuples in the head relation to be instantiated. This is done by sending messages addressed to the tuple host node. For broadcast, a special asterisk symbol "*" is used as the location specifier.

```
store(@*,Object) :- produce(@X,Object).
process(@X,Object) :- store(@X,Object).
```

The first rule broadcasts the store tuple. In the second rule, any neighboring node X that receives this broadcast replaces the location specifier of the tuple with its local id.

2.2.2 Interfacing to the Physical World

In order to link declarative programs to hardware such as sensors and actuators, users may specify *built-in relations*. For example, the prior example's thermometer relation may read values from the underlying temperature sensor.

```
builtin(temperature, 'TemperatureImplementor.c').
```

ThermometerImplementor.c is a external module (written in a language like nesC (43)) implementing the thermometer relation. This method of exposing sensors as tables is similar to TinyDB. Actuation is exposed similarly. Here, sounder tuples result in sound as implemented by the SounderImplementor.c.

```
builtin(sounder, 'SounderImplementor.c').
sounder(@Node,frequency) :- process(@Node,Object).
```

2.2.3 Querying for Data

Users pose *queries* to specify that certain relations are of interest and should be output from the DSN runtime. Queries are indicated by a tuple terminated by a question mark.

```
interestingRelation(@AllHosts,InterestingValue)?
```

When a new tuple of this type is generated, it is also transmitted to the user. Currently the Serial interface is used for this purpose. If no query is specified in a program, all the relations are considered of interest and delivered.

Additional `netlog` constructs will be presented in the following sections as needed. A comprehensive DSN tutorial is also available for interested programmers (44).

2.3 A Tour of Declarative Sensornet Programs

In this section, we investigate `netlog`'s potential for expressing core sensor network protocols, services and applications. Through a series of sample programs, we tackle different sensor network problems, at multiple, traditionally distinct levels of the system stack. For the sake of exposition, we will tend to explain `netlog` programs in rule-by-rule detail, though auxiliary statements like type definitions are elided from the listings. Complete program listings are availabe at (44)

2.3.1 Tree Routing: A Common Network Service

In-network spanning-tree routing is a well-studied sensor network routing protocol. Tree construction is a special case of the Internet's Distance Vector Routing (DVR) protocol. Nodes simply construct a spanning tree rooted at the base by choosing the node that advertises the shortest cost to the base as their next hop neighbor. This tree construction in `netlog` is presented in Listing 2.1.

Each node starts with only information about link qualities of neighboring nodes given by the relation link(@Host,Neighbor,Cost). For all nodes, the root of the tree is explicitly specified with a fact (line 2), and a bootstrap value for the shortest cost to the root is also set (line 3).

To establish network paths to the root, first nodes that are one hop neighbors from the root use local links to the destination as network paths to the root. This corresponds to the rule in line 5, which reads, "If a node Source wishes to reach a destination Dest and has a local link to this destination with cost Cost, then establish a network path path from Source to Dest with next hop of Dest and cost Cost."

The tilde symbol "∼", such as in this rule's dest relation, indicates that the arrival of new

```
1  % Initial facts for a tree rooted at ''root''
2  dest(@AnyNode, root).
3  shortestCost(@AnyNode, root, infinity).
4  % 1-hop neighbors to root (base case)
5  path(@Source, Dest, Dest, Cost) :- dest(@Source, Dest)~,
     link(@Source, Dest, Cost).
6  % N-hop neighbors to root (recursive case)
7  path(@Source, Dest, Neighbor, Cost) :- dest(@Source, Dest)~,
     link(@Source, Neighbor, Cost1),
     nextHop(@Neighbor, Dest, NeighborsParent, Cost2), Cost=Cost1+Cost2,
     Source!=NeighborsParent.
8  % Consider only path with minimum cost
9  shortestCost(@Source, Dest,<MIN, Cost>) :- path(@Source, Dest, Neighbor, Cost),
     shortestCost(@Source, Dest, Cost2)~, Cost < Cost2.
10 % Select next hop parent in tree
11 nextHop(@Source, Dest, Parent, Cost) :- shortestCost(@Source, Dest, Cost),
     path(@Source, Dest, Parent, Cost)~.
12 % Use a natively implemented link table manager
13 builtin(link, 'LinkTableImplementor.c').
```

Listing 2.1: Tree Routing

tuples from the associated body relation do not trigger the reevaluation of the rule. This is useful in the cases that reevaluation is unwanted or unnecessary.

Second, nodes that are more than one hop from the root establish paths by deduction. A node Source that has a neighbor Neighbor that already has a established a path to the root can construct a path that goes through this neighbor with a cost that is the sum of the link cost Cost1 to neighbor and the neighbor's cost to the root Cost2 (line 7).

Here, path tuples are possible paths to the root, whereas nextHop tuples are only the shortest paths to the root. The reduction of possible paths to the shortest path occurs in the two rules of line 9 and 11. We employ a MIN database aggregation construct over the set of possible paths to find the minimum cost.

After successful tree construction, each node has selected a parent with the least cost to

get to the destination. This information is captured in the nextHop(@Source,Dest,Parent,Cost) relation and represents the network-level forwarding table for all nodes.

So far we have glossed over how the local link table link is populated and maintained. For now, let us assume a link table manager provided by an external component (line 13). In Section 2.4, we discuss several reasonable alternatives to constructing this link table, including a link estimator constructed declaratively.

This program does not downgrade tree paths when link qualities decrease. We can additionally add this mechanism with three more rules.

Besides serving as data collection sinks, trees often serve as routing primitives (45; 46; 10). Construction of multiple trees based on this program is very easy; a second tree only requires the addition of two facts for the new root such as dest(@AnyNode,root2) and shortestCost(@AnyNode,root2, infinity).

2.3.2 Multi-hop Collection: An Initial User Application

To perform periodic multi-hop collection, we forward packets on top of tree routing at epoch intervals. This is very similar to a popular use-case of TinyDB (31). The program is shown in Listing 2.2.

We first import our previous tree routing such that we can use its nextHop forwarding table (line 2). The two built-ins used are timer for interacting with the hardware timer, and thermometer for reading the temperature (line 3 and 4).

A fact sets the initial timer (line 7). A timer relation on the right side (body) is true when the timer fires, and a timer relation on the left side (head) evaluating to true indicates the timer is being set. Therefore, having the same timer relation in the body and head creates a reoccurring timer (line 8). This timer's main purpose is to periodically sample temperature and initiate a multi-hop send (line 11).

Conceptually, multi-hop routing on a tree involves recursively matching a transported message with the appropriate forwarding tables along the path to the destination. This recursion

```
1
2  import('tree.snl').
3  builtin(timer, 'TimerImplementor.c').
4  builtin(thermometer, 'ThermometerImplementor.c').
5
6  % Schedule periodic data collection
7  timer(@AnyNode, collectionTimer, collectionPeriod).
8  timer(@Src, collectionTimer, Period) :- timer(@Src, collectionTimer, Period).
9
10 % Sample temperature and initiate multihop send
11 transmit(@Src, Temperature) :- thermometer(@Src, Temperature),
     timer(@Src, collectionTimer, Period).
12
13 % Prepare message for multihop transmission
14 message(@Src, Src, Dst, Data) :- transmit(@Src, Data),
     nextHop(@Src, Dst, Next, Cost)~.
15
16 % Forward message to next hop parent
17 message(@Next, Src, Dst, Data) :- message(@Crt, Src, Dst, Data),
     nextHop(@Crt, Dst, Next, Cost)~, Crt != Dst.
18
19 % Receive when at destination
20 receive(@Crt, Src, Data) :- message(@Crt, Src, Dst, Data), interest(@Crt, Data),
     Crt == Dst.
```

Listing 2.2: Multi-hop Collection

is expressed succinctly in an initialization rule (line 14), recursive case (line 17) and base case (line 20) above. The initialization rule prepares the application level send request into a generic message suitable for forwarding. The recursive case forwards the message (at either an originating or intermediate node) according to each recipient's nextHop entry for the final destination. Finally, upon receipt at the final destination, the message is passed upward to the application layer if it matches its interest (line 20).

The message relation takes the place of the standard network queue. As such, we are able

to design any queue admission policy through our operations on relations, such as unification and database-style aggregation. On the other hand, queue eviction policies are limited by the system-provided table eviction mechanisms. We discuss provisions for table eviction in Section 2.4.

2.3.3 Distributed Version Coherency: Translating From Pseudocode

Various sensor network protocols utilize a version coherency dissemination algorithm to achieve eventual consistency. Listing 2.3 illustrates a declarative implementation of a leading approach, version coherency with the Trickle dissemination algorithm. (32). Despite the algorithm's complexity, we were very pleasantly surprised by how easy it was to implement in `netlog`. In fact, the comments in Listing 2.3 are directly from the original Trickle paper pseudocode (32). Save for setting timers in lines 2-6, each line of pseudocode translates directly into one rule. This example in particular lends evidence to our claim that `netlog` is at an appropriate level of abstraction for sensor network programming.

The Trickle algorithm provides conservative exponential-wait gossip of metadata when there is nothing new (line 9), aggressive gossip when there is new metadata or new data present (lines 15 and 18), both counter-balanced with polite gossip when there are competing announcers (line 12). Underscores in a relation's arguments, such as in timer of line 9, represent "don't care" unnamed variables.

The algorithm is inherently timer intensive. Trickle's timer T, corresponding to tTimer in the listing, performs exponential-increase of each advertisement epoch. Timer τ, corresponding to tauTimer, performs jittered sending in the latter half of each epoch in order to avoid send synchronization. Lines 24 and 25 store and update to the new version once the new data is received.

```
1  % Tau expires: Double Tau up to tauHi. Reset C, pick a new T.
2  tauVal(@X,Tau*2) :- timer(@X,tauTimer,Tau), Tau*2 < tauHi.
3  tauVal(@X,tauHi) :- timer(@X,tauTimer,Tau), Tau*2 >= tauHi.
4  timer(@X,tTimer,T) :- tauVal(@X,TauVal), T = rand(TauVal/2,TauVal).
5  timer(@X,tauTimer,TauVal) :- tauVal(@X,TauVal).
6  msgCnt(@X,0) :- tauVal(@X,TauVal).
7
8  % T expires: If C < k, transmit.
9  msgVer(@*,Y,Oid,Ver) :- ver(@Y,Oid,Ver), timer(@Y,tTimer,_), msgCnt(@Y,C),
     C < k.
10
11 % Receive same metadata: Increment C.
12 msgCnt(@X,C++) :- msgVer(@X,Y,Oid,CurVer), ver(@X,Oid,CurVer), msgCnt(@X,C).
13
14 % Receive newer metadata: Set Tau to tauLow. Reset C, pick a new T.
15 tauVal(@X,tauLow) :- msgVer(@X,Y,Oid,NewVer), ver(@X,Oid,OldVer), NewVer >
     OldVer.
16
17 % Receive newer data: Set Tau to tauLow. Reset C, pick a new T.
18 tauVal(@X,tauLow) :- msgStore(@X,Y,Oid,NewVer,Obj), ver(@X,Oid,OldVer),
     NewVer > OldVer.
19
20 % Receive older metadata: Send updates.
21 msgStore(@*,X,Oid,NewVer,Obj) :- msgVer(@X,Y,Oid,OldVer),
     ver(@X,Oid,NewVer), NewVer > OldVer, store(@X,Oid,NewVer,Obj).
22
23 % Update version upon successfully receiving store
24 store(@X,Oid,NewVer,Obj) :- msgStore(@X,Y,Oid,NewVer,Obj).
     store(@X,Oid,OldVer,Obj), NewVer > OldVer.
25 ver(@X,Oid,NewVer,Obj) :- store(@X,Oid,NewVer,Obj).
```

Listing 2.3: Trickle Version Coherency

```
1  builtin(trackingSignal,'TargetDetectorModule.c').
2  import('tree.snl').

4  % On detection, send message towards cluster head
5  message(@Src,Src,Head,SrcX,SrcY,Val) :- trackingSignal(@Src,Val),
     detectorNode(@Src), location(@Src,SrcX,SrcY), clusterHead(@Src,Head).
6  message(@Next,Src,Dst,X,Y,Val) :- message(@Crt,Src,Dst,X,Y,Val),
     nextHop(@Crt,Dst,Next,Cost).
7
8  % At cluster head, do epoch-based position estimation
9  trackingLog(@Dst,Epoch,X,Y,Val) :- message(@Dst,Src,Dst,X,Y,Val),
     epoch(@Dst,Epoch).
10 estimation(@S,Epoch,<AVG,X>,<AVG,Y>) :- trackingLog(@S,Epoch,X,Y,Val),
     epoch(@S,Epoch).
11
12 % Periodically increment epoch
13 timer(@S,epochTimer,Period) :- timer(@S,epochTimer,Period).
14 epoch(@S,Epoch++) :- timer(@S,epochTimer,_), epoch(@S,Epoch).
```

Listing 2.4: Tracking

2.3.4 Tracking: A Second End-User Application

Listing 2.4 shows a multi-hop entity tracking application implemented in `netlog`. The specification is faithful to what has been presented in recently deployed tracking applications (34). The algorithm works as follows. A node that registers a detection via the trackingSignal sends a message to the cluster head indicating the position of the node (lines 5 and 6). The cluster head node periodically averages the positions of the nodes that sent messages to estimate the tracked object's position (line 10). To correctly compute the destination for each epoch, the trackingLog relation labels received messages with the estimation epoch in which they were received (line 9). Periodic timers update the current epoch (lines 14-13).

This application uses a fixed cluster head. Four additional rules can be added to augment the program to specify a cluster head that follows the tracked target.

2.3.5 Additional Examples

All of the preceding examples discussed in this section compile and run in DSN. We implemented and validated basic geographic routing (47), NoGeo localization (33) and exponentially weighted moving average link estimation (48) as well. These program appear in Appendix A. Additionally, we have sketched implementations of other sensor network services such as: in-network data aggregation (49), beacon vector coordinate and routing protocol BVR (45), data-centric storage protocol pathDCS (46), and geographic routing fallback schemes such as right hand-rules and convex hulls (50; 47). Our conclusion is that `netlog` implementations of these applications pose no fundamental challenges, being expressible in code listings of no more than several dozen rules while all running over the same minimal DSN runtime discussed in Section 2.5.

2.4 Beyond Expressing Sensornet Services

In the previous section, we showed that the declarative approach is natural for defining a wide range of sensornet services. In this section, we discuss two additional advantages. First, the declarative approach naturally accommodates flexible system architectures, an important advantage in sensornets where clear architectural boundaries are not fixed. Second, DSN facilitates resource management policies using simple declarative statements.

2.4.1 Architectural Flexibility

Disparate application requirements and the multitude of issues that cut across traditional abstraction boundaries, such as in-network processing, complicate the specification of a single unifying sensornet architecture: one size may not fit all. DSN strives to accommodate this need for architectural flexibility.

First, it is both possible and reasonable to declaratively specify the entire sensornet application and all supporting services, save for hardware device drivers. The previous section showed

specifications of both high-level applications such as tracking and intermediate services such as Trickle. In addition, we have specified cross-layer applications such as in-network aggregation (49) and low-level protocols such as link estimation. For link estimation, we implemented a commonly-used beaconing exponentially weighted moving average (EWMA) link estimator (48) in `netlog`, detailed in Appendix A. The combination of the link estimator with tree routing and multi-hop collection presented in Section 2.3 constitutes an end-user application written entirely in `netlog`, except for the hardware-coupled built-ins thermometer and timer.

At the same time, it is straightforward to adopt packaged functionality with built-in relations. For instance, we initially implemented link as a built-in, since it allows us to expose radio hardware-assisted link-estimations in lieu of our declarative link estimator. As a third option, we also used SP (51), a "narrow waist" link layer abstraction for sensor networks as a built-in. In the next subsection, we outline how a substantial system service, energy management, can be incorporated into declarative programs. Similarly, higher-level functionality implemented natively such as a network transport service can also be incorporated. In this way, DSN facilitates users who want to program declaratively while retaining access to native code.

Architectural flexibility in DSN is also attractive because relations can provide natural abstractions for layers above and below, such as link(Node,Neighbor,Cost) for the neighbor table and nextHop(Node,Destination,Parent,Cost) for the forwarding table. These relations' tuples are accessed just like any others, without special semantics assigned to their manipulation. We can also see similar intuitive interfaces in other instances: geographic routing also provides a nextHop forwarding table like tree routing; geographic routing and localization, which are naturally interrelated, use and provide the location(Node,X,Y) relation respectively, which is simply a table of each node's location. Both geographic routing and localization are presented in Appendix A. In these cases, the declarative approach facilitates program composition with intuitive abstraction interfaces.

Yet, the right level of declarative specification remains an open question. While a single sensornet architecture has not emerged to date, one may yet crystallize. By enabling users to

freely mix and match declarative programming with existing external libraries, DSN enables the future exploration of this subject.

2.4.2 Resource Management

As a consequence of the physical constraints of sensornet platforms, DSN offers flexibility in the management of three fundamental resources: memory, processor and energy.

2.4.3 Memory

Since current sensornet platforms are memory constrained, DSN makes several provisions for managing memory effectively. At the programming level, the user is able to specify: the maximum number of tuples for a relation, tuple admission and eviction strategies, and insertion conflict resolution policies. The materialize and typedef statements set these policies for a relation.

```
materialize(relationName, entryTimeout, maxEntries, evictPolicy).
typedef(#relationName, ^uint16_t, uint32_t, ^uint8_t).
```

The materialize statement sets a maximum number of entries for relationName and a timeout value for each tuple after which it will be removed. The eviction policy specifies how tuples should be removed when the maximum number of allocated tuples has been exceeded. Standard policies included in the runtime are random, least recently used and deny. This construct, borrowed from (30), permits the user to effectively specify static memory usage more simply than traditional sensornet programming systems.

The typedef statement specifies the relation's argument types, as well as properties of the relation's primary key. The primary key is indicated by the hat symbol "^" next to the arguments that are part of the primary key. In the above statement, the first and third argument comprise the primary key of relationName. Primary keys are useful for memory management because only one copy of a tuple of the same primary key is stored. The hash symbol "#" preceding relationName indicates that the policy is to replace old entries with new entries when

primary keys are equal. The absence of the hash symbol indicates that the policy is to keep old entries. Explicit materialize and typedef are often optional, because of reasonable defaults settings and program type inference.

Because we use a high-level language, the compiler has significant opportunity for optimization. For example, we have implemented two different memory layout schemes for generated DSN binaries, trading off between code and data memory. Since sensor network platforms separate code from data, *i.e.,* ROM from RAM, the compiler can optimize binary generation depending on the particular type of hardware platform. Section 2.6 discusses this more in depth. The combination of programming and compilation options enables a deductive database in a reasonable memory footprint.

2.4.4 Processor

Determining execution preferences among competing, possibly asynchronous, events is important, especially in embedded systems. For example, it may be desirable to prioritize event detection over background routing maintenance. DSN uses a priority mechanism to let the user specify tuple processing preference. For example, high temperature detection is prioritized over the rest of the processing below:

```
priority(highTemperature,100).

% Background rules
reportHumidity(...) :- ... .
disseminateValue(...) :- ... .

% Rule fired by prioritized relation
reportHighTemperature(...) :- highTemperature(...),... .
```

In the above example, if multiple new tuples in the system are ready to be processed, the highTemperature tuples will be considered for deductions first, before the other regular priority tuples.

Prioritized deduction offers a simple way for users to express processing preferences, without worrying about the underlying mechanism. It also differs from traditional deduction where execution preferences are not directly exposed to users.

Additionally, priorities can address race conditions that may arise when using intermediate temporary relations, since DSN does not provide multi-rule atomicity. These races can be avoided by assigning high priority to temporary relations.

2.4.5 Energy

Effective energy management remains a challenging task. Several systems have attempted to tackle this problem, such as Currentcy (52) for notebook computers, and a somewhat similar sensornet approach (53). These energy management frameworks provide (1) a user policy that allocates and prioritizes energy across tasks, and (2) a runtime energy monitor and task authorizer.

Since declarative languages have been previously used for policy, we wished to assess the suitability of adopting energy management into DSN. Below, we outline how `netlog` programs can naturally incorporate energy management.

For user policy, it is straightforward to specify relations concerning desired system lifetime, energy budgets for individual sensors, and resource arbitration of energy across system services. As one example, facts of the form em_PolicyMaxFreq(@host,actionId,frequency) set up maximum frequencies allowed by the energy manager for different actions.

For task authorization, checks to the energy accounting module occur as part of a rule's body evaluation. To do this, we make authorization requests by including a em_Authorize(@host,actionId) relation in the body of rules that relate to actionId. This means that these rules must additionally satisfy the authorization check to successfully execute.

The two new relations mentioned map fairly naturally to the native energy management interface envisioned by the authors in (53) and (52). Listing 2.5 provides an example of an `netlog` program with these energy management features.

```
1  % Energy Manager-specific relations
2  builtin(em_Authorize, EMModule).
3  builtin(em_PolicyMaxFreq, EMModule).

5  % Permit light actions at most 10 times per minute
6  em_PolicyMaxFreq(@Src,lightAction,10).
7
8  % Permit temperature actions at most 20 times per minute
9  em_PolicyMaxFreq(@Src,temperatureAction,20).
10
11 % Log light readings
12 lightLog(@Src,Reading) :- photometer(@Src,Reading),
     em_Authorize(@Src,lightAction).
13
14 % Sample temperature readings and send them to the base
15 temperatureReport(@Next,Reading) :- thermometer(@Src,Reading),
     nextHop(@Src,Dst,Next,Cost), em_Authorize(@Src,temperatureAction).
```

Listing 2.5: Specifying Energy Policy

The energy-aware program specified in Listing 2.5 stores light readings locally, and forwards temperature samples to a base station. Different policies are associated with each of the two main actions, lightAction and temperatureAction. (lines 6 and 9). Authorization for lightAction is requested when logging light readings, while the request for temperatureAction is processed when sampling and sending the temperature readings (line 12 and 15 respectively). If the energy budget is depleted, the underlying EMModule will evaluate these requests in accordance to the specified user policy.

In addition to addressing energy, a vital resource constraint in sensornets, this exercise also demonstrates flexibility in incorporating a new system service into DSN's existing architecture. The DSN programming tutorial provides further assistance and examples for interfacing DSN with native system services (44).

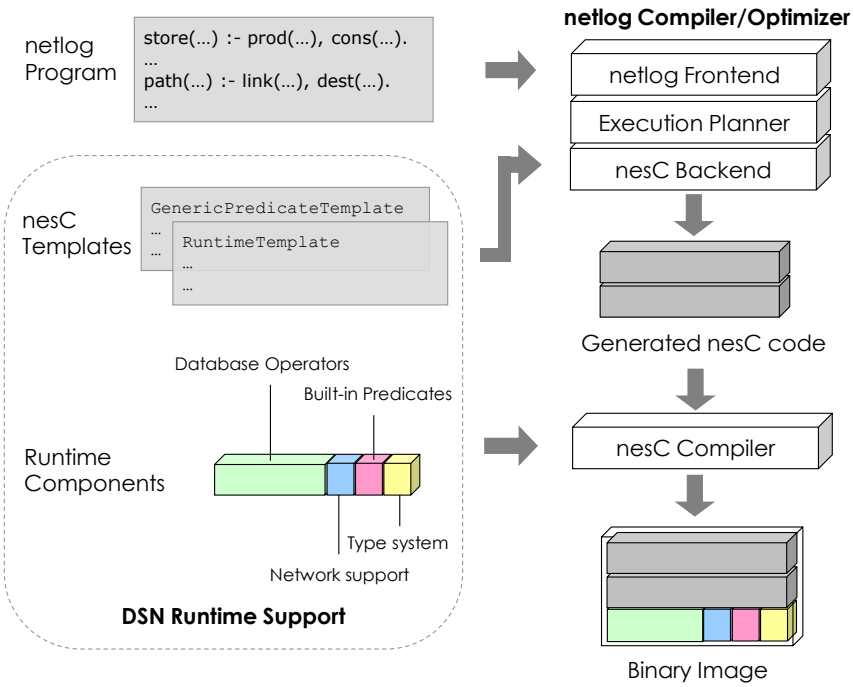


Figure 2.1: DSN Architecture. `netlog` is compiled into binary code and distributed to the network, at which point each node executes the query processor runtime.

2.5 System Architecture

In this section we present a high level view of our system design and implementation. The high level architecture for transforming `netlog` code into binary code that runs on motes is shown in Figure 2.1. At the core of the framework lies the `netlog` compiler that transforms the `netlog` specification into the nesC language (43) native to TinyOS (54). The generated components, along with preexisting compiler libraries, are further compiled by the nesC compiler into a runtime implementing a minimal query processor. This resulting binary image is then programmed into the nodes in the network.

As an overview, each rule from the `netlog` program gets transformed in the compiled code into a sequence of components that represent database operators like join, select, and project, which, to facilitate chaining, implement uniform push/pull interfaces. The runtime daemon

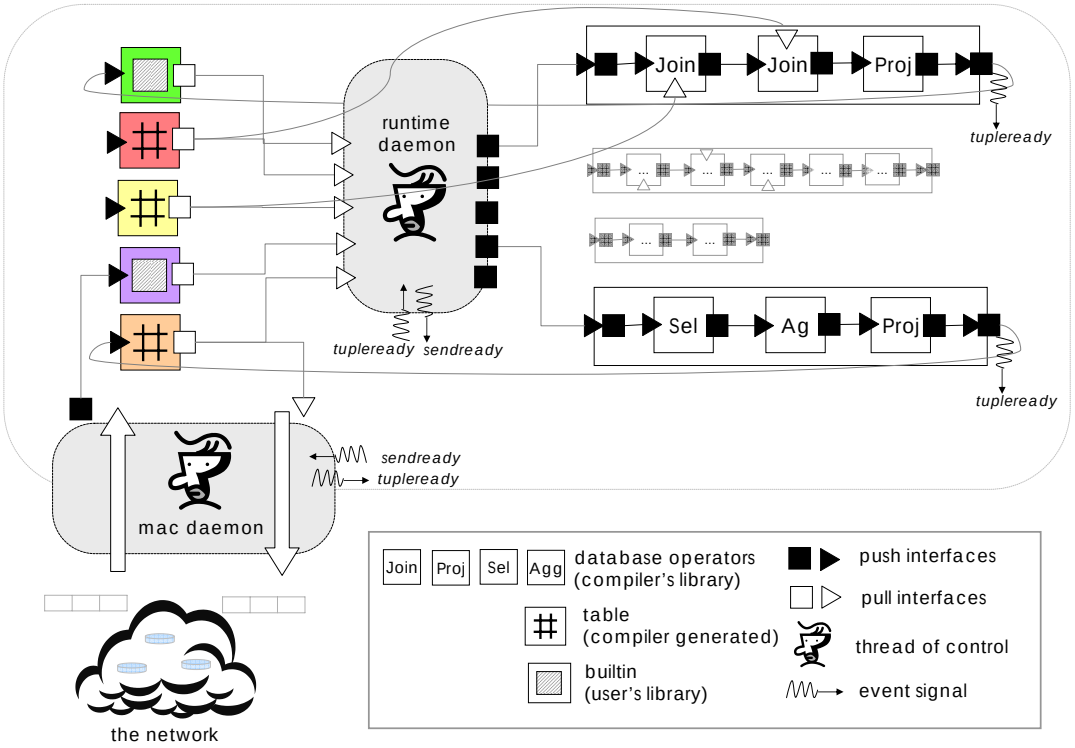


Figure 2.2: DSN Runtime. Each rule is compiled into a dataflow chain of database operators.

manages the dataflow processing of tuples from and to tables and built-ins while the network daemon manages tuples arriving from and destined to the network. Figure 2.2 presents an overall view of this runtime activity.

2.5.1 The Compiler

A fundamental choice of DSN is heavy use of PC-side program compilation as opposed to mote-side program interpretation. This relates directly to our goals of reducing runtime memory footprint and providing predictable operation.

The compiler parses the `netlog` program and does a set of initial rule-level level transformations on distributed rules (those whose location specifiers are not all the same). Next, it translates the program into an intermediary dataflow representation that uses chains of

database operators (such as joins and selects) to describe the program. Then, for each chain, the compiler issues nesC code by instantiating components from a set of compiler library generic templates. Finally, the generated components, the system runtime and any necessary library runtime components are compiled together into a binary image using the nesC compiler.

2.5.2 The Runtime

We chose to implement the runtime system as a compiled dataflow of the user provided rules in the `netlog` program. As is well known in the database community, declarative logic maps neatly to dataflow implementations. An example compiled runtime is shown in Figure 2.2.

The constrained resources and predictability concerns of sensor nodes make full fledged query processors for our purposes (*e.g.,* runtime rule interpreters) difficult to justify. While interpreters are used in several high-level sensor network languages for data acquisition (40; 31), we were wary of the performance implications of interpreting low-level services such as link estimators. In addition, we felt static compiler-assisted checks prior to deployment were worth any loss of flexibility. As a result of aggressive compilation, the resulting runtime system is equivalent to a dedicated query processor compiled for the initial set of rules, allowing new tuples (but not new rules) to be dynamically inserted into the network.

2.5.3 Code Installation

We rely on traditional embedded systems reprogrammers to distribute initial binary images onto each node prior to deployment. Users are free to install different rules and facts on different nodes, while retaining a common relation set definition (database schema) across nodes. This permits basic support for different nodes requiring different functionality, as in heterogeneous sensor networks.

2.6 Implementation

In this section we discuss implementation design decisions and detail compiler and runtime interactions.

2.6.1 Implementation Choices

In the following, we explain the most important implementation choices we made and how they affect the performance and semantics of the system. The resulting system exhibits sizeable dissimilarities from existing networked deductive databases (30).

2.6.2 Dynamic vs Static allocation

TinyOS does not have a default dynamic memory allocation library. On the other hand, database systems often make substantial use of dynamic memory allocation, and previous systems like TinyDB (31) have implemented dynamic memory allocation for their own use. In our implementation, we decided to use static allocation exclusively. While dynamic allocation may better support the abstractions of limitless recursion and flexible table sizes, static allocation remained preferable for the following reasons. First, we believe that static allocation with a per-relation granularity gives programmers good visibility and control over the case when memory is fully consumed. By contrast, out-of-memory exceptions during dynamic allocation are less natural to expose at the logic level, and would require significant exception-handling logic for even the simplest programs. Second, our previous experiences indicated that we save a nontrivial amount of code space in our binaries that would be required for the actual dynamic allocator code and its bookkeeping data structures. Finally, because tuple creation, deletion and modification of different sizes is common in DSN, the potential gains of dynamic allocation could be hard to achieve due to fragmentation. Instead, in our system all data is allocated at compile time. This is a fairly common way to make embedded systems more robust and predictable.

2.6.3 Memory Footprint Optimization

In general, in our implementation we chose to optimize for memory usage over computation since memory is a very limited resource in typical sensor network platforms, whereas processors are often idle.

Code vs. Data Tradeoff: Our dataflow construction is convenient because, at a minimum, it only requires a handful of generic database operators. This leads to an interesting choice on how to create instances of these operators. *Code-heavy generation* generates (efficient) code for every operator instance, whereas *data-heavy generation* generates different data parameters for use by a single generic operator. This choice affects the sizes and ratios of code and data memory of the generated binary. Many microprocessors common in current sensor nodes present strict boundaries between code and data memory (*i.e.,* ROM vs. RAM). The choice is further influenced by the volatile/nonvolatile characteristics of the different memory modules (*e.g.,* typically only ROM is persistent, holding both code and data constants). We have implemented both modes of parameter generation. For our primary platform TelosB (55), it typically makes sense to employ data-heavy generation because of the hardware's relative abundance of RAM. However, for other popular platforms that DSN supports, the reverse is true. The choice ultimately becomes an optimization problem to minimize the total bytes generated subject to the particular hardware's memory constraints. Currently this decision is static and controls dataflow operators in a program, but in principle this optimization could be automated based on hardware parameters.

Reduce Temporary Storage: To further improve memory footprint, we routinely favored recomputation over temporary storage. First, unlike many databases, we do not use temporary tables in between database operators but rather feed individual tuples one at a time to each chain of operators. Second, all database operator components are implemented such that they use the minimal temporary storage necessary. For instance, even though hash joins are computationally much more efficient for evaluating unifications, our use of nested loop joins avoids any extra storage beyond what is already allocated per relation. Our aggregation withholds use of traditional group tables by performing multiple table scans on inputs. Finally, when passing parameters between different components, we do not pass tuples but rather generalized tuples, *Gtuples*, containing pointers to the already materialized tuples. Gtuples themselves are caller-allocated and the number necessary is known at compile time. The use of Gtuples saves significant memory space and data copying, and is similar to approaches in traditional databases (56).

2.6.4 Rule Level Atomicity

In our environment, local rules (those whose location specifiers are all the same) are guaranteed to execute atomically with respect to other rules. We find that this permits efficient implementation as well as convenient semantics for the programmer. In conjunction with rule level atomicity, priorities assist with execution control and are discretionary rather than mandatory. In addition, by finishing completely the execution of a rule before starting a new rule we avoid many potential race conditions in the system due to the asynchronous nature of relations (*e.g.,* tuples received on the network) and to the fact that we share code among components.

2.6.5 Implementation Description

Below we present more details on the DSN system implementation such as component interactions and the network interface. We call a "table" the implementation component that holds the tuples for a relation.

2.6.6 Compiler

Frontend and Intermediary The frontend is formed by the following components: the lexical analyzer; the parser; the high level transformer and optimizer (HLTO); and the execution planner (EP). The parser translates the rules and facts into a list which is then processed by the HLTO, whose most important goal is rule rewriting for distributed rules. The EP translates each rule into a sequence of database operators. There are four classes of operators our system uses: Join, Select, Aggregate and Project. For each rule, the execution planner generates several dataflow join plans, one for each of the different body relations that can trigger the rule.

Backend nesC Generator The nesC Generator translates the list of intermediary operators into a nesC program ready for compilation. For each major component of our system we use template nesC source files. For example, we have templates for the main runtime task and each of the operators. The generator inserts compile-time parameters in the template files, and also generates linking and initialization code. Examples of generated data are: the

number of columns and their types for each relation, the specific initialization sequences for each component, and the exact attributes that are joined and projected. Similarly, the generator constructs the appropriate mapping calls between the generated components to create the desired rule.

2.6.7 Runtime Interactions

Our dataflow engine requires all operators to have either a push-based open/send/close or pull-based open/get/close interface. The runtime daemon pushes tuples along the main operator path until they end up in materialized tables before operating on new tuples, as in Figure 2.2. This provides rule-level atomicity. To handle asynchrony, the runtime daemon and network daemon act as pull to push converters (pumps) and materialized tables act as push to pull converters (buffers). This is similar to Click (57).

A challenging task in making the runtime framework operate correctly is to achieve the right execution behavior from the generic components depending on their place in the execution chain. For instance, a specific join operator inside a rule receiving a Gtuple has to pull data from the appropriate secondary table and join on the expected set of attributes. A project operator has to know on which columns to project depending on the rule it is in. Furthermore, function arguments and returns must be appropriately arranged. To manage the above problem under data-heavy generation, we keep all necessary data parameters in a compact parse tree such that it is accessible by all components at runtime. The component in charge of holding these parameters is called *ParamStore*. The task of ensuring the different operational components get the appropriate parameters is done by our compiler's static linking. Under code-heavy generation, we duplicate calling code multiple times, inlining parameters as constants.

2.6.8 Built-in Relations

Well-understood, narrow operator interfaces not only make it very easy to chain together operators, but also facilitate development of built-in relations. In general, users can write

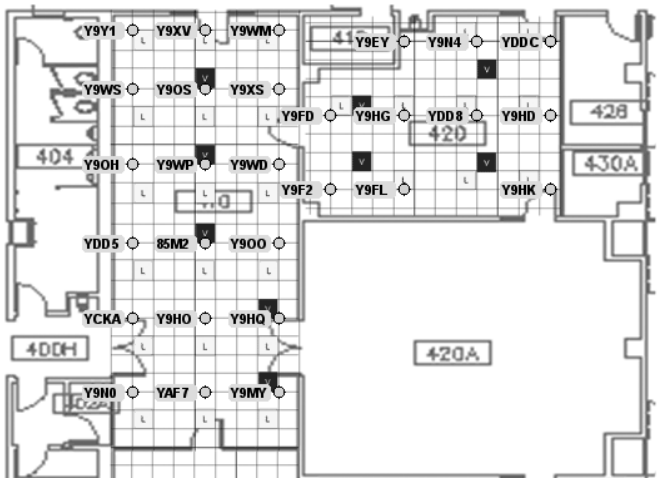


Figure 2.3: 28 mote Omega Testbed at UC Berkeley

arbitrary rules containing built-in relations and can also include initial facts for them. Some built-ins only make sense to appear in the body (sensors) or only in the head (actuators) of rules, while others may be overloaded to provide meaningful functionality on both the head and body (*e.g.,* timer). We permit this by allowing built-ins to only provide their meaningful subset of interfaces.

2.7 Evaluation

In this section we evaluate a subset of the `netlog` programs described in Section 2.3. We analyze DSN's behavior and performance in comparison with native TinyOS nesC applications using a 28 node testbed (58) shown in Figure 2.3 and TOSSIM (59), the standard TinyOS simulator.

2.7.1 Applications and Metrics

We present evaluations of tree formation, collection, and Trickle relative to preexisting native implementations. Furthermore we describe our experience in deploying a DSN tracking application at a conference demo.

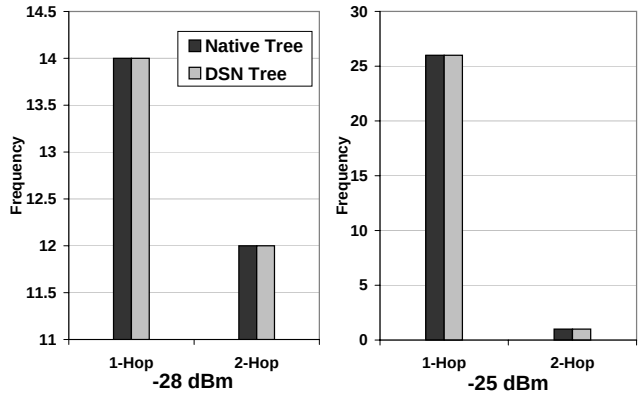
Three fundamental goals guide our evaluation. First, we want to establish the correctness of the `netlog` programs by demonstrating that they faithfully emulate the behavior of native implementations. Second, given the current resource-constrained nature of sensor network platforms, we must demonstrate the feasibility of running DSN on the motes. Finally, we perform a quantitative analysis of the level of effort required to program in `netlog`, relative to other options.

To demonstrate the correctness of our system, we employ application-specific metrics. To evaluate tree-formation, we look at the distribution of node-to-root hop-counts. We then run collection over the tree-formed by this initial algorithm, measuring end-to-end reliability and total network traffic. For Trickle, we measure the data dissemination rate as well as the number of application-specific messages required. To demonstrate feasibility, we compare code and data sizes for `netlog` applications with native implementations. Finally, we use lines of code as a metric for evaluating ease-of-programming.

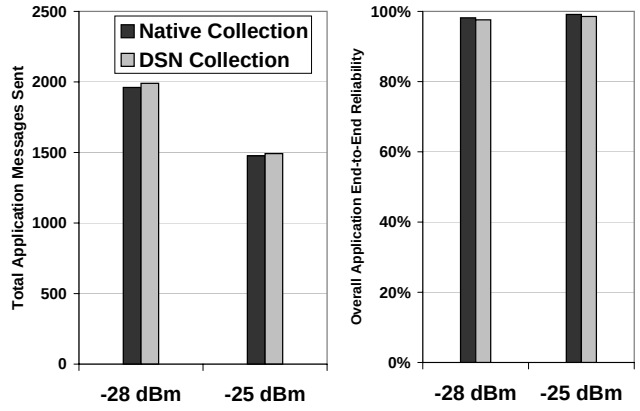
2.7.2 Summary of Results

The results indicate that DSN successfully meets algorithmic correctness requirements. DSN Tree forms routing trees very similar to those formed by the TinyOS reference implementation in terms of hop-count distribution and our collection implementation achieves nearly identical reliability as the native implementation. Finally, DSN Trickle provides near-perfect emulation of the behavior of the native Trickle implementation.

In terms of feasibility, DSN implementations are larger in code and data size than native implementations. However, for our profiled applications, our overall memory footprint (code + data) is always within a factor of three of native implementation and all our programs fit



(a) Hop count distribution to the root at two power levels.



(b) Total application messages transmitted and overall network end-to-end reliability for collection.

Figure 2.4: Tree-formation and collection on the 28 node testbed.

within the current resource constraints. Additionally, several compiler optimizations which we expect will significantly reduce code and data size are still unimplemented.

Concerning programming effort, the quantitative analysis is clear: the number of lines of nesC required for the native implementations are typically orders of magnitude greater than the number of rules necessary to specify the application in `netlog`. For example tree construction requires only 7 rules in `netlog`, as opposed to over 500 lines of nesC for the native implementation.

2.7.3 Tree/Collection Correctness Tests

For tree formation, we compared our DSN Tree presented in Section 2.3 to MultihopLQI, the de facto Native Tree implementation in TinyOS for the Telos platform. To compare fairly to Native Tree, we augmented DSN Tree to perform periodic tree refresh using the same beaconing frequency and link estimator. This added two additional rules to the program.

To vary node neighborhood density, we used two radio power levels: power level 3 (-28dBm), which is the lowest specified power level for our platform's radio, and power level 4 (-25dBm). Results higher than power level 4 were uninteresting as, given our testbed, the network was entirely single-hop. By the same token, at power level 2, nodes become partitioned and we experienced heavy variance in the trials, due to the unpredictability introduced by the weak signal strength at such a power level.

Figure 2.4a shows a distribution of the frequency of nodes in each hop-count for each implementation. As a measure of routing behavior, we record the distance from the root, in terms of hops, for each node. Node 11, the farthest node in the bottom left corner in Figure 2.3 was assigned the root of the tree. We see that both DSN Tree and Native Tree present identical distributions at both power levels.

The collection algorithm for DSN, presented in Section 2.3, runs on top of the tree formation algorithm discussed above. For testing the Native Collection, we used TinyOS's SurgeTelos application, which periodically sends a data message to the root using the tree formed by the underlying routing layer, MultihopLQI. Link layer retransmissions were enabled and the back-channel was again used to maintain real-time information.

Figure 2.4b shows the results of the experiments for two metrics: overall end-to-end reliability, and total message transmissions in the network. The network-wide end-to-end reliability of the network was calculated by averaging the packet reception rate from each node at the root. We see that DSN Collection and Native Collection perform nearly identically, with an absolute difference of less than 1%.

	DSN Trickle	Native Trickle
Total Messages Sent	299	332
Suppressed Messages	344	368

Table 2.1: Trickle Messages

2.7.4 Trickle Correctness Tests

In order to demonstrate that the `netlog` version of Trickle presented in Section 2.3 is an accurate implementation of the the Trickle dissemination protocol, we compare the runtime behavior of our implementation against a widely used native Trickle implementation, Drip (60). To emulate networks with longer hopcounts and make a more precise comparison, we performed the tests in simulation rather than on the previous two hop testbed. Data is gathered from simulations over two grid topologies of 60 nodes: one is essentially linear, arranging the nodes in a 30×2 layout and the other is a more balanced rectangular 10×6 grid. The nodes are situated uniformly 20 feet apart and the dissemination starts from one corner of the network. We used lossy links with empirical loss distributions.

Figure 2.5 presents simulation results for the data dissemination rate using the two implementations. These results affirm that the behavior of the DSN and the native implementation of Trickle are practically identical.

In addition, we counted the total number of messages sent by the two algorithms and the number of message suppressions. Table 2.7.4 presents the total number of Trickle messages sent by both implementations and the total number of suppressed messages for the 30×2 topology. Again, these results demonstrate the close emulation of native Trickle by our DSN implementation.

2.7.5 Tracking Demo

We demonstrated the tracking application specified in `netlog` (and presented in Section 2.3) at a conference (61). Our set-up consisted of nine TelosB nodes deployed in a 3×3 grid with the communication range set such that each node only heard from spatially adjacent neighbors.

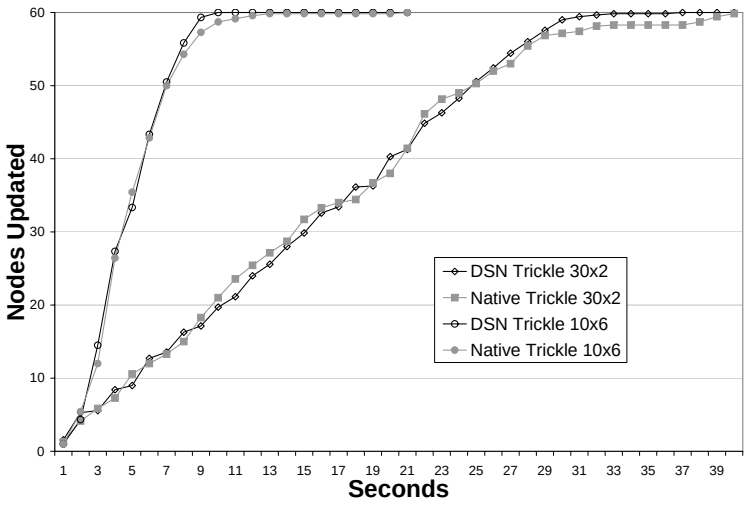


Figure 2.5: Trickle dissemination rate in Tossim simulation.

A corner-node base station was connected to a laptop, which was used for displaying real-time tracking results and up-to-date network statistics collect from the network. A tenth "intruder" node broadcasted beacon messages periodically and the stationary nodes then tracked the movement of this intruder and reported their observations to the base station. The demo successfully highlighted the specification, compilation, deployment, and real-time response of a tracking application similar to actually deployed tracking applications (34).

2.7.6 Lines of Code

Measuring the programmer level of effort is a difficult task, both because quantifying such effort is not well-defined and a host of factors influence this effort level. However, as a coarse measure of this programming difficulty, we present a side-by-side comparison of the number of lines of nesC native code against the number of lines of `netlog` logic specifications necessary to achieve comparable functionality. This approach provides a quantifiable metric for comparing the level of effort necessary across different programming paradigms.

Table 2.7.6 provides a comparison in lines of code for multiple (functionally equivalent) implementations of tree routing, data collection, Trickle and tracking. The native version

Program	Lines of Code			
	Native	NLA	TinyDB	DSN
Tree Routing	580	106	-	7 Rules (14 lines)
Collection	863	-	1	12 Rules (23 lines)
Trickle	560	-	-	13 Rules (25 lines)
Tracking	950	-	-	13 Rules (34 lines)

Table 2.2: Lines of Code Comparison

refers to the original implementation, which is currently part of the TinyOS distribution (54). NLA, or network layer architecture, is the implementation presented in (62), which decomposes sensornet network protocols into basic blocks. It is not fit for expressing non-routing services.

The reduction in lines of code when using `netlog` is dramatic at roughly two orders of magnitude. TinyDB is also extremely compact, consisting of a single line query. However, as Section 5.1 discusses, TinyDB is limited to only data acquisition, rather than entire protocol and application specification. We conjecture that such a large quantitative distinction translates into a qualitatively measurable difference in programming effort level. To this we also add our subjective (and admittedly biased) views that during the development process, we strongly preferred programming in `netlog`, as opposed to nesC.

2.7.7 Feasibility

In this section we evaluate the feasibility of our system to meet the hard memory constraints of the current sensor network platforms. We show that there is a significant fixed cost for our runtime system, but this is manageable even for the current platforms and comparable to existing proposals.

2.7.8 Code/Data size

The TelosB mote, the main platform on which DSN was tested, provides 48KB of ROM for code, and 10KB of RAM for data.[1] Given these tight memory constraints, one of our initial concerns was whether we could build a declarative system that fits these capabilities.

Table 2.7.8 presents a comparison in code and data size for the three applications profiled in Table 2.7.6. For a fair comparison, the presented memory footprints for the native applications do not include modules offering extra functionality which our implementation does not support. Note however that the extracted modules still have a small impact on the code size due to external calls and links to/from them.

Program	Code Size (KB)			Data Size (KB)		
	Native	NLA	DSN	Native	NLA	DSN
Tree Routing	20.5	24.8	24.8	0.7	2.8	3.2
Collection	20.7	-	25.2	0.8	-	3.9
Trickle	12.3	-	24.4	0.4	-	4.1
Tracking	27.9	-	32.2	0.9	-	8.5

Table 2.3: Code and Data Size Comparison

The main reason for the larger DSN *code size* is the size of the database operators. As an important observation, note that this represents a *fixed* cost that has to be paid for all applications using our framework. This architectural fixed cost is around 21kB of code and 1.4kB of data. As we can see in Table 2.7.8, constructing bigger applications has only a small impact on code size.

On the other hand, the main reason for which the DSN *data size* is significantly larger than the other implementations is the amount of parameters needed for the database operators and the allocated tables. This is a variable cost that increases with the number of rules, though, for all applications we tested, it fit the hardware platform capabilities. Moreover, although not yet implemented, there is significant room for optimization and improvement in our compiler

[1]The Mica family of platforms are also supported but compiler optimizations favorable to the Micas are not implemented.

backend. Finally, if data size were to become a problem, the data memory can be transfered into code memory by generating more operator code and less operator parameters (see Section 2.6).

The overall *memory footprint* (measured as both code and data) of DSN implementations approaches that of the native implementations as the complexity of the program increases. Such behavior is expected given DSN's relatively large fixed cost, contrasted with a smaller variable cost.

We also mention that our system is typically more flexible than the original implementations. For instance, in the collection tree routing implementation, we are able to create multiple trees with the addition of two `netlog` initial fact, and no additional code (unlike the native implementation).

As a final note, technology trends are likely to produce two possible directions for hardware: sensor nodes with significantly more memory (for which memory overhead will be less relevant), and sensor nodes with comparably limited memory but ever-decreasing physical footprints and power consumption. For the latter case, we believe we have proved by our choice of Telos platform and TinyOS today that the overheads of declarative programming are likely to remain feasible as technology trends move forward.

2.7.9 Execution Overhead

Two additional potential concerns in any system are network packet size overhead and runtime delay overhead. Our system adds only a single byte to packets sent over the network, serving as an internal relation type identifier for the tuple payload. Finally, from a runtime delay perspective, we have not experienced any delays or timer related issues when running declarative programs.

2.8 Limitations

We divide the current limitations of our approach into two categories. First, there are certain drawbacks that are inherent to a fully declarative programming approach, which we have only been able to ameliorate to a degree. Second, DSN has certain limitations. The restrictions that fall into the second category can typically be lifted by introducing additional mechanisms/features; we leave these for future work. Conversely, while the shortcomings of the declarative approach can potentially be mitigated, they still remain as fundamental costs of declaratively specifying systems.

As noted in Section 2.1.1, a declarative language hides execution details that can be of interest to the programmer. This has two implications. First, it is less natural to express some programming constructs where imperative execution order is required. Second, the declarative approach is not appropriate for code with very high efficiency requirements such as low level device driver programming. For instance, in our declarative program, the granularity of user control is the rule; the user cannot preempt rule execution. This also implies that real time requirements may be hard to guarantee in the declarative system. Therefore, we expect the low level programming for device drivers to be done natively and incorporated through built-ins.

Going one step further, we observe that while the high level language offers more room for compiler optimizations, the overall efficiency of a system implemented declaratively will most likely not surpass a painstakingly hand-tuned native one. Fundamentally, we are trading expressivity and programming ease for efficiency, and this may be the right tradeoff in a variety of scenarios.

Finally, a declarative sensor network system has to interface with the outside world, and the callouts to native code break the clean mathematical semantics of deductive logic languages. In this case there is some potentially useful prior work on providing semantic guarantees in the face of such callouts (63).

A few of the limitations of DSN were briefly discussed in Section 2.1.1, namely the ability to do only polynomial-time computation and the lack of support for complex data objects. These

are somewhat ameliorated by the ability of DSN to call out to native code via built-in predicates. While the computational complexity restraint will not likely affect DSN's practicality, the lack of complex data objects may. One might consider implementation of an Abstract Data Type system akin to that of Object-Relational databases, to enable more natural declarations over complex types and methods (64). In addition, we recognize that many embedded programmers may be unfamiliar with `netlog` and its predecessor Datalog. We actively chose to retain `netlog`'s close syntactical relationship to its family of deductive database query languages, though certainly more familiar language notations may facilitate adoption.

Currently, users can only select among a fixed set of eviction policies for tables. We are considering a language extension which would allow users to evict based on attribute value, a construction that we expect to fit most practical eviction policies.

Finally, in Section 2.4.2 we presented several mechanisms to increase the user control over the execution, such as with priorities to express preference for the tuple execution order. We note that these constructs take the expressive power of our language outside the boundaries of traditional deductive database semantics, and a formal modeling of these constructs remains an important piece of future work.

2.9 Summary

Data and communication are fundamental to sensor networks. Motivated by these two guiding principles, we have presented a declarative solution to specify entire sensor network system stacks. By example, we showed several real `netlog` programs that address disparate functional needs of sensor networks. These programs' text were often orders of magnitude fewer lines of code, yet still matched the designer's intuition. In addition, DSN enables simple resource management and architectural flexibility by allowing the user to mix and match declarative and native code. This lends considerable support to our hypothesis that the declarative approach may be a good match to sensor network programming. The DSN system implementation

shows that these declarative implementations are faithful to native code implementations and are feasible to support on current sensor network hardware platforms.

Chapter 3

Rendezvous and Proxy Selection

3.1 Motivation

Declarative languages traditionally offer ease and compactness of expressiveness, as well as separation of policy from mechanism. Having found DSN to be a good fit for expressing a mutlitude of sensornet programs, we next investigate opportunities to automatically optimize execution. This is analogous to how standard SQL statements are optimized by a general-purpose database optimizer before execution. The key difference is the challenges posed by networked systems. We focus on optimizations that address prototypical networking rendezvous and proxy placement concerns.

- Where should messages from communicating parties rendezvous? Should the rendezvous occur via push, pull or some of both?
- Who should hold the conversation state of an ongoing communication?
- Should applications send (application) data to routers, or conversely should routers send (routing) data to applications? Is a mixture of each the most appropriate?

These concerns are not limited to sensornets, but often arise in traditional networked systems as well. System builders often wrestle with these choices in concrete instances to achieve better performance. Section 3.1.1 discusses some of these situations in detail. Our optimizer, `netopt`, mitigates the need for case-by-case consideration of rendezvous and proxy selection. This is timely in the networking environment, given increasingly diverse and varying workloads and resources.

In addition to the utility of our optimizations, a conceptual contribution of our work is in exposing the congruence between network design and recursive query optimization, a traditional topic in database theory. Specifically, we show that optimal network rendezvous and proxy selection are analogous to cost-based selection pushing in the presence of recursive queries.

To examine the utility of our network optimizations, we apply them to both traditional networking and sensornet settings. In simulation, gains are by as much as two orders of magnitude. On testbeds, gains are by as much as one order of magnitude. In both settings, `netopt` effectively identifies and executes better strategies.

This chapter is organized as follows. Section 3.1.1 takes a closer look at our two chosen application scenarios. Section 3.2 introduces our distributed and recursive query language and offers initial attempts at network optimization. Section 3.3 discusses the main rendezvous optimization in detail. Section 3.4 extends this to proxy placement optimization. Section 3.5 discusses their execution on our implementation platforms. Section 3.6 reports on prototype implementation and deployment of our optimizations. Section 3.8 summarizes this chapter's results.

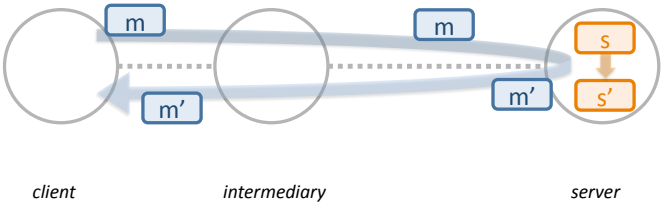
3.1.1 Two Networking Settings

This section takes a closer look at two networking settings: wireless sensing and content distribution. In each, we pay attention to aspects where manual rendezvous and proxy placement decisions have significantly impacted system design.

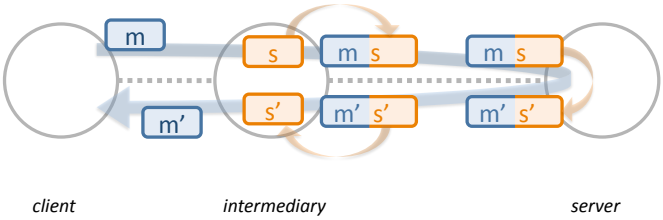
Sensornets

One predominant application class for sensornets is event detection and distribution. In the naive variant, an event source sends event notifications to the sink *i.e.,* rendezvous only at the sink. Yet rendezvous at other locations in the network is conceptually possible and often beneficial. For example, if many events are generated but only a few are of interest to the sink, it may be more energy efficient to pass the sink's selection criteria (part of the way) to the source. Furthermore, in the case of multiple source and sink pairs, limited node buffer space may preclude every pair from using its preferred rendezvous. The optimization problem is akin to ones encountered outside sensornets *e.g.,* in Pub-Sub (65) and Content Distribution Networks (21) where it is known to correspond to the NP-complete facilities location problem (15). We show how `netopt` can automatically identify naive cases from program source, rewrite them to expose rendezvous flexibility, and assign lower cost rendezvous.

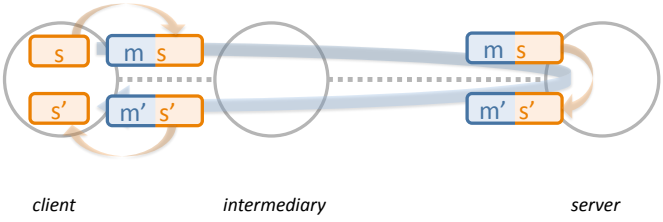
Recently, many common Internet services such as interactive login, remote debugging and point to point routing are being ported to sensornets. A challenge that arises repeatedly is that of configuring state allocation on storage-constrained platforms. Such state varies in form and use, from interactive login sessions to routing table entries. Should it reside at either endpoint, at intermediate proxies, or in packets? And who makes these decisions? These alternatives are illustrated in Figure 3.1. The service designer implementing something like interactive login will not know the needs and constraints of a each specific deployment. On the other hand, the end system deployer can not be expected to be intimately familiar with reconfiguring the protocols of every packaged service. Unfortunately, this implies that neither is in the best position to optimize state allocation. The result is that conservative service designers minimize node state at the expense of increasing in-flight packet state. Since the radio is frequently the most power-intensive hardware unit, the increased communications directly decrease the overall network lifetime. We tackle this problem with automated techniques for exposing and optimizing proxy placement.



(a) "Stateful" execution



(b) State at Proxy execution



(c) "Stateless" execution

Figure 3.1: Three alternate executions for client-server communication. m represents the message from client to server. Upon reaching the server, m is modified to m'. s represents the session state held at the server on behalf of the client-server communication. It is also modified upon m reaching the server.

Content Distribution Networks

Website operators that wish to offer more responsive sites often turn to Content Distribution Networks (CDNs) such as Akamai and Limelight (66; 67). CDNs host third party content

on their large server collections, ideally placing popular items close to interested consumers. Optimal event notification and content distribution are known to be related problems. Therefore, optimizer-driven rendezvous selection applies to both sensornet and CDN settings.

Each client that connects to a server allocates its own session state, as shown in Figure 3.1a. As the number of clients increases, server responsiveness may suffer due to session state exhausting available server memory. To alleviate the problem, session state can be repackaged into messages for transportation between client and server, allowing the server to be stateless, as shown in Figure 3.1c (19; 68). Similarly, as a hybrid alternative, session state packaged in messages can be picked up from and dropped off at an intermediate proxy on the path connecting client and server, as shown in Figure 3.1b. This proxy placement problem is complementary to rendezvous selection, and in the same mold as the proxy placement problem described for sensornets.

3.2 Example Program Optimization

As a basis for optimization, we continue to employ `netlog`. This section revisits an example application, and probes an initial attempt to expose more rendezvous choices for the application. A main tool used throughout the optimizations, *network selection pushing*, is also introduced.

3.2.1 An Initial Program

Listing 3.1 revisits the core event distribution logic introduced earlier in Chapter 2.3, and is very similar to Listing 2.2. It implements multi-hop message forwarding from sources to sinks with message filtering at sinks. We shall call this program `BasicProg`. The data is routed via the second rule of `BasicProg` by recursively defining the contents of the *message* relation with respect to the *nexthop* relation. Intuitively, *message* tuples are traversing the *nexthop* routing tables (lines 7-9). Upon arrival at the sink, the *message* tuple, if it matches any tuples in *interest*, generates *consume* tuples at the destination via the third rule of `BasicProg` (lines 12-14). The query indicates that asks for the *consume* queried relation (line 17).

```
1  % Prepare for transmission
2  message(@Source, Source, Sink, Data) :-
3    produce(@Source, Data),
4    nexthop(@Source, Sink, Next).
5
6  % Route message to next hop parent
7  message(@Next, Source, Sink, Data) :-
8    message(@Current, Source, Sink, Data),
9    nexthop(@Current, Sink, Next).
10
11 % Receive if message is of interest
12 consume(@Sink, Data) :-
13   message(@Sink, Source, Sink, Data),
14   interest(@Sink, Data).
15
16 % What is consumed?
17 consume(@Sink, Data)?
```

Listing 3.1: Original `BasicProg`, event distribution from source to sink with filtering by interest.

3.2.2 Pushing Selections One-Hop

As an example, consider a two node network x and y represented by a previously-defined set of facts (sometimes called an "Extensional Database (EDB)" (69)) D consisting of three relations:

```
produce(@y, foo).  nexthop(@y, x, x).
interest(@x, foo).
```

The EDB is the set of relations that are never in the head of any rule; its tuples are defined exogenously, perhaps via a data structure in a persistent store. Conversely, the Intensional Database (IDB) is made of the derived relations that occur in rule heads. The IDB of D is:

```
message(@y, y, x, foo).  message(@x, y, x, foo).
consume(@x, foo).
```

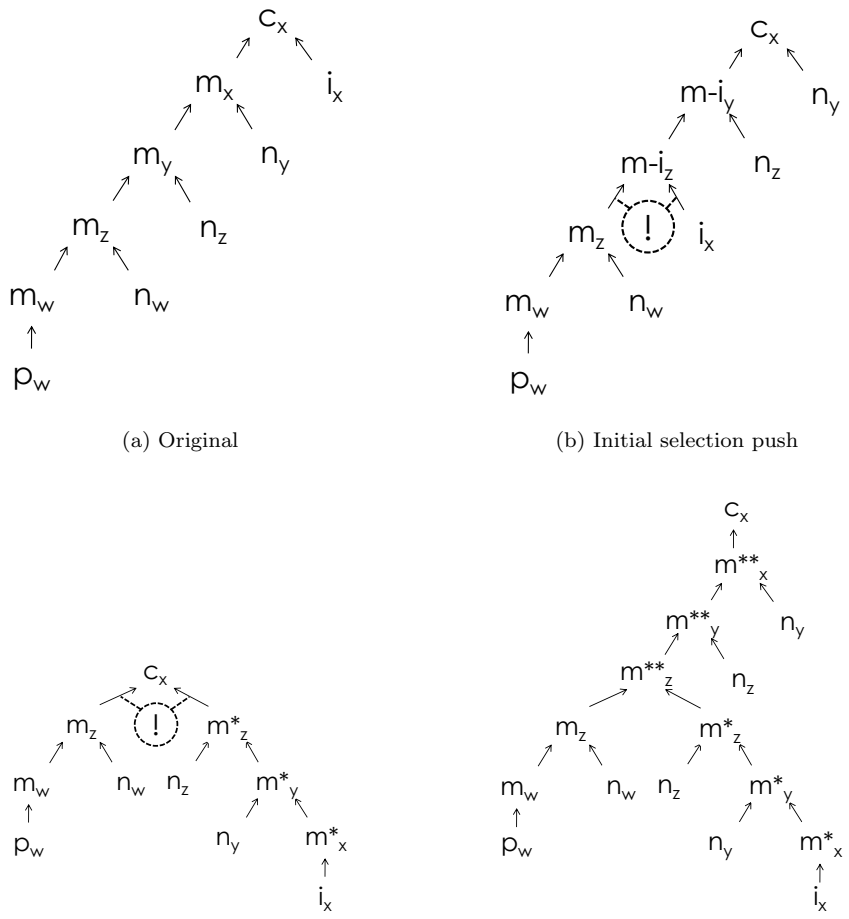
In `BasicProg`, *produce* and *interest* rendezvous at a node x by sending *message* from some node y to x. Conceptually, this rendezvous could also take place at y as long as the query returns the same answer, $consume(@a, foo)$. To accomplish this, let *interest* send its own "message" from x to y. We'll call it *message*$*$, and use it in the following rules:

```
message*(@Current, Current, Data) :-
  interest(@Current, Data).

message*(@Current, Sink, Data) :-
  message*(@Next, Sink, Data),
  nexthop(@Current, Sink, Next).

consume(@Sink, Data) :-
  produce(@Current, Data),
  message*(@Current, Sink, Data).
```

The first rule prepares *interest* tuples as *message*$*$ tuples. The second rule passes *message*$*$ *backward* along *nexthop*, and is similar to how *message* was routed in Listing 3.1. The third rule derives *consume*. For the one-hop network, these rules produce the desired result of rendezvous at y, with the queried *consume* at x. As a result, we have "pushed" the selection condition *i.e., interest* back to *produce*.

3.2.3 Pushing Selections into the Network

As the network topology grows to multiple hops, we would like to add a bit more flexibility to this rewrite attempt. At the moment, we must choose between either endpoint, which is similar to a technique mentioned in (29). In a multi-hop network, rendezvous at any intermediary hop should be an option. We next provide some intuition on how network selection pushing generalizes to the multi-hop case. A program's network execution can be visualized with a *network derivation graph*. Figure 3.2a shows the network derivation graph for `BasicProg` over a four hop linear network with nodes x-y-z-w. Each network derivation graph node ρ_ξ represents a horizontal partition of relation ρ at location ξ (relation names are abbreviated



(a) Original

(b) Initial selection push

(c) Path-restricted *interest*

(d) Fully path-restricted

Figure 3.2: Alternative executions of `BasicProg`. Exclamation marks indicate neighboring hosts are not connected in the network topology.

by their first letter). A directed edge leads from derivation input to derivation output. For example, n_z represents the rows of *nexthop* that are stored at location z, and the edge from p_w to m_w indicates that the program derives *message* at w from *produce* at node w. A node with a fan-in greater than one indicates a join among the node's children, as in the case of m_z and the join of m_w and n_w.

We can push selections to achieve a different network execution. Figure 3.2b shows the

network derivation graph resulting from an initial selection push. Here, the join of *message* and *interest* is performed earlier, resulting in subsequent *message* tuples already filtered by *interest* (denoted $m - i$). Conceptually, the "pushing down" of *interest* changes rendezvous of *message* and *interest* from x to z. However, x and z are not neighbors in the underlying network topology (as indicated by the exclamation mark). Hence, they cannot communicate directly with each other and the partitions $interest_x$ and $message_z$ cannot directly join. In general, `netlog` programs require the following property for proper distributed execution.

Definition 3.2.1. *A rule is **path-restricted** if all head and body relation partitions are located on the same host or neighboring hosts in the underlying network topology. A program is path-restricted if its rules are path-restricted.*

We assume that input programs are path-restricted, and we would like to maintain the property for any rewritten programs. Figure 3.2c suggests an alternate join rearrangement that is path-restricted for $interest_x$. It is roughly the result of combining Listing 3.1 with the rules in Section 3.2.2. $produce_w$ is converted to *message* and travels from w to z, $interest_x$ is converted to $message*$ and travels from x to y to z. This leaves $message*_z$ and $message_z$ ready to join at z.

However, the derivation of $consume_x$ involves z and x that are not neighbors. To resolve this issue, we can "package up" $consume_x$ as a new relation $message**$ and send it along the network topology via a path we already know about from z to x. Figure 3.2d shows this as part of the fully path-restricted network derivation graph with rendezvous at z. This is just one possible rendezvous choice. The "Meet-in-the-Middle" *MiM Rewrite* we discuss next transforms input programs to expose many possible rendezvous choices.

3.3 Meet-in-Middle Rewrite

The `netopt` network optimization architecture executes in three stages:

- *Analysis* identifies optimization opportunities from input programs. We show how to identify rendezvous and proxy selection opportunities.
- *Rewriting* primes programs for optimization by transforming input programs to optimizable variants.
- *Decision Making* selects optimized configurations. The optimizer installs its chosen configuration by simply filling in tables initialized by *Rewriting* to list selected rendezvous and proxies.

This section first sets forth the correctness criteria of any `netopt` optimization, and describes the MiM Rewrite procedure precisely in terms of its analysis and rewrite phases. Any `netopt` optimization must preserve the intent of the original program. The intent is captured by the query.

Definition 3.3.1. *Two programs P_1, P_2 are **query equivalent** if, given any EDB, the contents of their queried relations are equivalent.*

Definition 3.3.2. *A rewriter $R : P_1 \rightarrow P_2$ is **query preserving** if for all programs P_1, P_2 is query equivalent to P_1.*

Note that neither of these definitions constrains the contents of the IDB in general, only the queried relations.

3.3.1 Analysis

Analysis identifies certain rules and relations as rewrite components. We first introduce some terminology from classic work in the deductive database literature (69).

Definition 3.3.3. *A **rule-goal graph** contains one relation-node for each relation and one rule-node for each rule. A directed edge leads from rule-node R to relation-node a if the head of rule R is relation a. A directed edge leads from relation-node a to rule-node R if relation a is in the body of rule R.*

Definition 3.3.4. *A rule with head relation a is a* ***linearly recursive rule*** *(LR rule) if a appears exactly once in the body. It is an* ***initializer rule*** *if a does not appear in the body.*

Definition 3.3.5. *A program is a* ***linearly recursive program*** *(LR program) if every rule with head a is (1) either an initializer rule or LR rule, and (2) for every relation b in the body, $b \neq a$, relation-node b in the rule-goal graph is not reachable from relation-node a.*

Definition 3.3.6. *A relation a is an* ***LR relation*** *if a is the head of an LR rule. The other relations in the body of the LR rule are* ***base relations****.*

Without loss of generality, we can restrict our discussion to scenarios in which the LR rule body contains only one base relation.[1] Our focus on networking programs leads us to consider the following type of LR rule.

R_1 $\mathtt{a(@b}_i\mathtt{,d}_1\mathtt{,\ldots,d}_{Na-1}\mathtt{)}$ $:-$ $\mathtt{a(@a}_1\mathtt{,\ldots,a}_{Na}\mathtt{)}$, $\mathtt{b(@b}_1\mathtt{,\ldots,b}_{Nb}\mathtt{)}$.

In the rule, the value b_i determines the new location specifier. Therefore, the partition of the head a is potentially different from the partition of the body a upon every recursion. Hence, we can interpret the base relation b as defining a network for the LR relation a to "hop along". For `BasicProg`, *message* is the only LR relation. *nexthop* is a base relation of *message*. Both LR and base relation identification can be accomplished by traversing the rule-goal graph.

Looking at the attribute variables in the example, note that each d_i can correspond to any a_i or b_i to get data values from the input to the output. Furthermore, b_i's can correspond to a_i's to capture join conditions between a and b.

Lastly, we are only interested in recursive relations that can (possibly indirectly) derive the queried relation because only they can impact query equivalency.

Definition 3.3.7. *Given queried relation c and LR relation a, a rule R is an* ***answer rule*** *if (1) a is in the body of R but not the head, and (2) in a rule-goal graph traversal, rule-node R can reach relation-node c.*

[1] When this is not so, it is straightforward to rewrite the program to include a rule that derives a single base relation by joining multiple base relations.

Given a program and queried relation, *Analysis* identifies LR and base relations, and LR, initializer and answer rules.

3.3.2 Rewriting

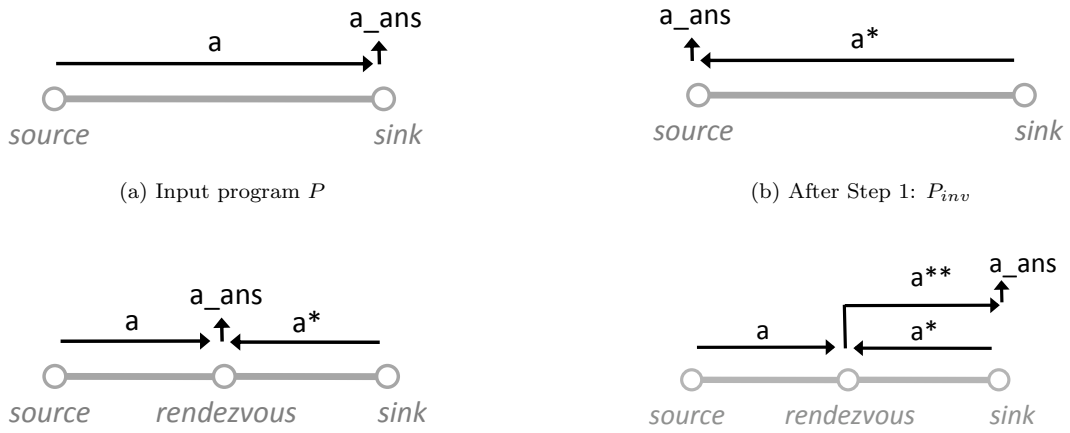
Using the rules and relations identified in *Analysis*, *Rewriting* invokes the MiM Algorithm. The MiM Algorithm transforms LR program P to a query equivalent program P_{MiM}. The advantage of P_{MiM} over P is that its rendezvous can be tuned by *Decision Making* by filling in tuples for a special *rendezvous* relation.

A preliminary procedure of the MiM Algorithm, common in the deductive database literature (70), canonicalizes the input program. First, recursive relation a is renamed a_ans in every answer rule for a. Second, for each rule, each variable is renamed to a unique variable name that does not appear elsewhere in the program (this is also known as "skolemization"). Third, a *binding list* for each recursive relation a is produced. A binding list α is a sequence of "b"s (bound) and "f"s (free), with each character representing an attribute of a. An attribute of a is bound ("b") if possible values are (1) already known since they are join keys with EDB relations, and (2) useful since the join happens in an answer rule. Informally, the binding list is a template that guides the search for derivations that might actually matter to the queried relation. For ease of exposition, we describe the algorithm only as it applies to the following type of answer rule where c is the head and e is in the EDB.

R_0 $\mathtt{c(@c_1, \ldots, c_{Nc})}$:- $\mathtt{e(@e_1, \ldots, e_{Ne})}$, $\mathtt{a(@a_1, \ldots, a_{Na})}$.

In this basic yet common case, the binding list α is assigned as follows: α_i is "b" if a_i joins with some e_j. Otherwise α_i is "f". Furthermore, with some trivial variable reordering, we can safely assume that α is a sequence of "b"s followed by a sequence of "f"s.

With these preliminaries, the core MiM Algorithm in Listing 3.2 is invoked. The shorthand notation it uses allows us to present MiM Algorithm compactly as a series of rule manipulations and variable list rearrangements. A term with a bar ("$^-$") represents a list of variables, and consists of a letter and optionally a digit, *e.g.,* $\overline{a1}$. The letter indicates that the size of the list



(a) Input program P

(b) After Step 1: P_{inv}

(c) After Step 2: P_{hyb}

(d) After Step 3: P_{MiM}

Figure 3.3: Steps of MiM Algorithm

is the number of attributes of the corresponding relation *e.g.,* $\overline{a1}$ is a variable list of size Na. The digit is just an identifier.

To manipulate variable lists, we use three functions. *unique* takes as input a list size and returns as output a list of distinct variables that do not appear anywhere else in any rule. *boundlist* takes as input a variable list for a and returns the prefix of the input for which α is "b". Conversely, $freelist$ returns the suffix of the input for which α is "f".

Each variable list originates from either (1) the input program P or (2) the function *unique*. Lastly, a term may have a subscript "b" or "f" to represent the application of the function *boundlist* or $freelist$ respectively. For example, $\overline{a1_b} = boundlist(\overline{a1})$. In such case, the length of $\overline{a1_b}$ may be less than that of $\overline{a1}$.

The MiM Algorithm generates new rules and introduces new relations a_ans, $a*$ and $a**$. In networking settings, tuples of a, $a*$ and $a**$ can be thought of as messages. Each message consists of a message header (some prefix of attributes) and message payload (remaining suffix of attributes). The header may change on every recursion but the payload does not.

The MiM Algorithm consists of three main steps traced by Figure 3.3. Figure 3.3a shows the input program as an *abstract network derivation graph* in which messages of a flow from

INPUT: A LR input program P with a binding list α for each recursive relation a. Recursive rules for a take the form:

R_1 a($\overline{a1}$) :- a($\overline{a2}$), b($\bar{b}$).

OUTPUT: An output program P_{MiM} having all the rules of P, with additional EDB relation r (rendezvous) and with each rule R_1 replaced by rules $R_{1.1}$, R_2, $R_{3.1}$, $R_{4.2}$, R_5 and R_6 as defined below.

PROCEDURE:

1. *Invert recursion order.* Generate P_{inv}, a version of P that processes derivations via "pull" rather than "push". The rules of P_{inv} are the rules of P with each rule R_1 replaced by three rules:

   R_2 a$*$($\overline{a0_b}$, $\overline{a0_b}$) :- ... *% answer rule dependent, refer to text*
   R_3 a$*$($\overline{a2_b}$, $\overline{a0_b}$) :- a$*$($\overline{a1_b}$, $\overline{a0_b}$), b($\bar{b}$).
   R_4 a_ans($\overline{a0_b}$, $\overline{a3_f}$) :- a$*$($\overline{a3_b}$, $\overline{a0_b}$), a($\overline{a3}$).
   with $\overline{a0_b}$ = ... *% answer rule dependent, refer to text*
   and $\overline{a3} = unique(Na)$.

2. *Hybridize recursion order.* Generate P_{hyb} by combining P_{inv} and P. In addition, add rendezvous relation r and modify selected rules to:

   a. Limit derivations of the queried relation to the rendezvous point. Replace R_4 with:

      $R_{4.1}$ a_ans($\overline{a0_b}$, $\overline{a3_f}$) :- a$*$($\overline{a3_b}$, $\overline{a0_b}$), a($\overline{a3}$), r($\overline{a3_b}$).

   b. Limit "push" execution to before the rendezvous and limit "pull" execution to after the rendezvous. Replace R_1 and R_3 with:

      $R_{1.1}$ a($\overline{a1}$) :- a($\overline{a2}$), b($\bar{b}$), −r($\overline{a2_b}$).
      $R_{3.1}$ a$*$($\overline{a2_b}$, $\overline{a0_b}$) :- a$*$($\overline{a1_b}$, $\overline{a0_b}$), b($\bar{b}$), −r($\overline{a1_b}$).

3. *Localize for network processing.* Generate P_{MiM} by modifying P_{hyb} to ensure network topology path restrictions. This enables correct distributed execution. Replace rules $R_{4.1}$ with:

   $R_{4.2}$ a$**$($\overline{a3_b}$, $\overline{a0_b}$, $\overline{a3_f}$) :- a$*$($\overline{a3_b}$, $\overline{a0_b}$), a($\overline{a3}$), r($\overline{a3_b}$).
   R_5 a$**$($\overline{a1_b}$, $\overline{a0_b}$, $\overline{a3_f}$) :- a$**$($\overline{a2_b}$, $\overline{a0_b}$, $\overline{a3_f}$), b($\bar{b}$).
   R_6 a_ans($\overline{a0_b}$, $\overline{a3_f}$) :- a$**$($\overline{a0_b}$, $\overline{a0_b}$, $\overline{a3_f}$).

Listing 3.2: MiM Algorithm

source to sink. After arriving at the sink, a generates a_ans, which participates in answer rules (not shown). More precisely, sources are locations where initializer rules generate a, and sinks are locations where answer rules use a.

Step 1 inverts the recursive order of the original program. Its objective is the same as to

that of the Magic Sets algorithm from database theory (71): pushing selection past recursion. This is done by constructing $a*$ to recurse backward from sink to source (Figure 3.3b). In networking terms, pushing down selections in this setting can be thought of as a sink-initiated "pull" execution vs. the original source-initiated "push" execution.

In Step 1 of Listing 3.2, R_2 is underspecified and we complete its specification here. Recall that given a queried relation c, α tells us that some attributes of a are already bound to specific values in an EDB relation. These are simply copied over to make $a*$, a superset of a. In the case of example R_0, R_2 takes the form:

$$\mathtt{a}*(\overline{a0_b}, \overline{a0_b}) \ :- \ \mathtt{e}(\overline{e}). \ \ \mathtt{with} \ \ \overline{a0} = \ @\mathtt{a}_1 \ , \ldots , \mathtt{a}_{Na} \ \ \mathtt{of} \ \ R_0$$

Note that two copies of the join keys are made. The first copy is like a message header that may need to go through some number of recursive modifications to find its join partners. The latter "pristine copy" is like a message payload with a return address, used to remember the original join keys for the answer rules.

Step 2 hybridizes the recursion order by combining push and pull execution to "meet-in-the-middle". It further introduces the EDB relation r whose tuples indicate the precise rendezvous meeting point between push and pull. While a traverses forward from source and $a*$ traverses backward from sink, both stop at the rendezvous point to derive a_ans (Figure 3.3c). Whereas P_{inv} pushes selection past recursion, P_{hyb} pushes selection into a tunable middle point in the recursion.

Step 3 localizes the program for network processing by ensuring that topology paths are respected. Essentially, a_ans is additionally packaged as a payload in another message, $a**$, and sent from rendezvous to sink. Upon reaching the sink, a_ans is unpackaged and can be used in answer rules, just as in the original program.

Steps 1 and 2 are applicable to any LR Datalog program. Step 3 is necessary for `netlog` programs that are expected to run on networks of nodes. We next present an example application of MiM Algorithm.

```
1  % Prepare for transmission
2  message(@Source, Source, Sink, Data) :-
3    produce(@Source, Data),
4    nexthop(@Source, Sink, Next).
5
6  % Route message to next hop parent until rendezvous
7  message(@Next, Source, Sink, Data) :-
8    message(@Current, Source, Sink, Data),
9    nexthop(@Current, Sink, Next),
10   -rendezvous(@Current, Sink, Data).
11
12 % Route interest back along next hop until rendezvous
13 message*(@Current, Current, Data) :-
14   interest(@Current, Data).
15 message*(@Current, Orig, Data) :-
16   nexthop(@Current, Sink, Next),
17   message*(@Next, Orig, Data),
18   -rendezvous(@Next, Sink, Data).
19
20 % At rendezvous, join message and interest and send to Sink
21 message**(@Current, Sink, Data) :-
22   message(@Current, Src, Sink, Data),
23   message*(@Current, Sink, Data),
24   rendezvous(@Current, Sink, Data).
25 message**(@Next, Sink, Data) :-
26   message**(@Current, Sink, Data),
27   nexthop(@Current, Sink, Next).
28 consume(@Sink, Data) :-
29   message**(@Sink, Sink, Data).
30
31 % What is consumed?
32 consume(@Sink, Data)?
```

Listing 3.3: Rewritten `BasicProg`, message and interest meet in the middle.

3.3.3 Example Application of MiM Algorithm

Listing 3.3 illustrates the result of a full application of the MiM Algorithm on `BasicProg`, with some variable renaming for ease of exposition. Appendix B shows the rewrite before variable renaming. In the rewritten `BasicProg` of Listing 3.3, the precise rendezvous location is chosen by simply filling in the *rendezvous* relation *e.g.,* with $rendezvous(@b, a, foo)$. The original recursion of *message* along *nexthop* is amended to include a negated term, $-rendezvous$ which modifies the interpretation of message routing to be: "Route *message* along *nexthop* until encountering *rendezvous*" (line 10). A similar negated term is applied to the routing back of *interest* (line 18). Additionally, MiM Rewrite amends `BasicProg` to deliver *consume* tuples in a multi-hop fashion to the *Sink* (lines 21-29) according to network path restrictions mentioned earlier.

Note that we have not specified nor constrained the tuples in the *rendezvous* relation. The decision of what to put there will be the task of *Decision Making*, discussed in Section 3.5. Next, we establish the correctness of MiM Rewrite by proving the following theorem.

Theorem 3.3.8. *The MiM Rewrite is query preserving and path-restricted.*

3.3.4 MiM Rewrite Correctness Proof

To prove this Theorem, we need some preliminary definitions and results. the structure of our discussion mirrors Listing 3.2.

[1. Inverting Recursion Order] We need the following constraint to show query equivalency of P_{inv} and P.

Constraint 1 (Free Variable Constraint)**.** *For each recursive rule having head a, the free variables of a in the head must be the same as the free variables of a in the body.*

This constraint says that the free variables of a are carried along unmodified from source to sink. Conversely, the bound variables of a may change upon every recursion. By analogy to networking, free variables are message payloads and bound variables are message routing headers. Provided this constraint, we borrow results from (69) (specifically Theorem 15.1) to claim that:

Lemma 3.3.9. *P and P_{inv} are query equivalent.*

Informally, we have already discussed how the construction of R_2 causes $a*$ to be filled with an initial superset of join key values for a starting at sink. R_3 takes the current set and asks what prior set is necessary to derive the current set, much like a depth first search from sink to source. Recall also that R_2 and R_3 store a "pristine copy" of the initial set, $\overline{a0_b}$, in the latter half of $a*$. Like a semi-join (72), (a subset of) $a*$ may eventually join with a at the source by R_4.[2] If this happens, since the free variables $\overline{a3_f}$ do not change with recursion (due to the constraint), they can be copied over directly from a to a_ans. Similarly, $a*$ copies its pristine copy $\overline{a0_b}$ to a_ans. The result is a_ans, which can now be used by any answer rule since free variables for a_ans have now been assigned values. The proof of Lemma 3.3.9 is by induction on the length of the path from source to sink.

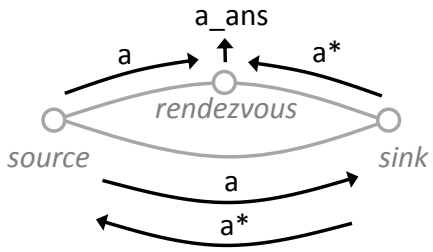
[2. Hybridizing Recursion Order] We can simultaneously enact push and pull processing by combining P and P_{inv}. However, this naive hybrid program causes many unnecessary derivations of identical a, $a*$ and a_ans tuples because neither push nor pull can detect when it has started duplicating the other's work; a goes completely from source to sink, $a*$ goes completely from sink to source, and at each point along the path, the same a_ans tuples are (re)derived. Limiting redundant derivations is not necessary for correctness but is preferable especially when redundant derivations lead to network communications overhead.

First, to limit redundant a_ans derivations, we name a particular rendezvous in a special rendezvous relation r. In Step 2.a, *Rewriting* adds r to the body of R_4, creating $R_{4.1}$. *Decision Making* populates the tuples of the r relation. It may choose any r as long as it obeys the following constraint.

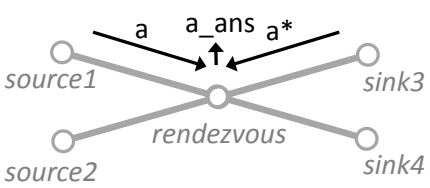
Constraint 2 (Selection Constraint)**.** *Any source and sink pair that share at least one path must have at least one rendezvous on one of the shared paths.*

Figure 3.4a shows an example where this constraint is respected. There are two paths from source to sink and at least one path, the top one, includes a rendezvous. No derivations of

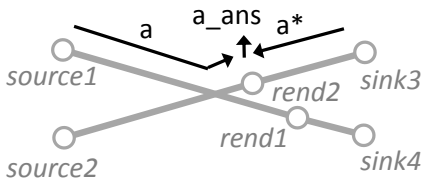
[2]Recall that initializer rules can also generate a.



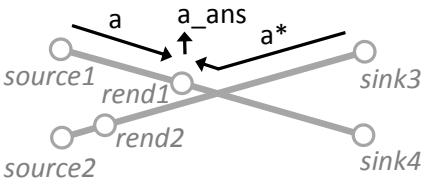
(a) Selection Constraint OK



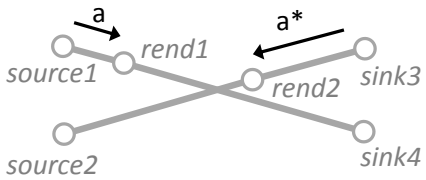
(b) Branching Constraint OK



(c) Branching Constraint OK



(d) Branching Constraint OK



(e) Branching Constraint BAD

Figure 3.4: MiM Algorithm constrains the location of rendezvous.

a_ans occurs on the bottom path due to failure to rendezvous *i.e.,* no entry of r lies on the bottom path.

Second, to limit redundant a and $a*$ derivations, we add r to the body of R_1 and R_3, creating $R_{1.1}$ and $R_{3.1}$. Here, r is negated, which means a and $a*$ traverse from source and sink respectively until encountering a rendezvous, but no further. To maintain correctness, *Decision Making* respects the following constraint in addition to Constraint 2.

Constraint 3 (Branching Constraint)**.** *Any single path between source and sink can have at most one rendezvous.*

Figures 3.4b-3.4e show three examples respecting this constraint and one example violating

this constraint. The setting is one in which $source1$ and $source2$ both have paths to $sink3$ and $sink4$. Such path "branches" at an intersection are analogous to network multicasts/one-to-many communication. Figure 3.4e is a violation because a from $source1$ and $a*$ from $sink3$ do not rendezvous at the same place. Provided Constraint 2 and 3, we have query equivalency after Step 2.

Lemma 3.3.10. *P and P_{hyb} are query equivalent.*

Proof. First consider P_{hyb} without the negated r terms. Since r's $\overline{a3_b}$ is a subset of a's $\overline{a3}$, $R_{4.1}$ cannot derive a_ans tuples that were not derived by R_4 (soundness). For completeness, proceed by induction on the choice of rendezvous. For the base case, choose the sink as the rendezvous, in which case P_{hyb} simplifies to P. For the induction, we can assume that query equivalency holds were we to pick the rendezvous one step closer to the sink at point $i+1$ rather than the current rendezvous choice at point i. By Lemma 3.3.9, LR relation a at $i+1$ implies a at i. Apply R_3 to $a*$ at $i+1$ to derive $a*$ at i. Finally apply $R_{4.1}$ to a, $a*$ and r (all at i) to derive a_ans. Constraint 2 ensures there is at least some rendezvous between source to sink for the inductive step.

Next consider P_{hyb} with the negated r terms. The addition of a negated term cannot make more derivations than without the negated term (soundness). Constraint 3 ensures that one path's rendezvous choice does not "block" rendezvous along another path, as seen in Figure 3.4e (completeness). □

[3. Restricting Derivations to Network Paths] $R_{4.1}$ is not a path-restricted rule. From the perspective of Figure 3.3c, a_ans is derived at the rendezvous but answer rules are expecting to use it at the sink. This is not an issue in a centralized Datalog execution. However, since the location specifier horizontally partitions relations in `netlog`, P_{hyb} needs path-restricting.

Path-Restricting Subprocedure modifies $R_{4.1}$ to ensure that head and body location specifiers are the same or are neighbors. It accomplishes this by providing hop-by-hop delivery of a_ans from body location to head location. Conceptually, Path-Restricting Subprocedure is a

generalization of link-restricted rewrites first proposed in (73) from the one-hop/non-recursive rules to multi-hop networks/recursive rules.

Specifically, Path-Restricting Subprocedure prepends "message header" attributes to a_ans as new relation $a{*}{*}$ (akin to encapsulation used in network tunneling). The message header is copied directly from a since it already had the appropriate header to get to the sink. This converts $R_{4.1}$ to $R_{4.2}$. Next, Path-Restricting Subprocedure constructs R_5 to let $a{*}{*}$ mimic a's multi-hop delivery. Lastly, R_6 unpackages $a{*}{*}$ into a_ans upon detecting that the outer message header is the same as the inner message header.

Finally, we claim Theorem 3.3.8.

Theorem 3.3.11. *The MiM Rewrite is query preserving and path-restricted.*

Proof. From Lemma 3.3.10 we already know P and P_{hyb} are query equivalent. To see that P and P_{MiM} are also query equivalent, observe that the message payload in $a{*}{*}$ representing a_ans is always copied and never modified. To see that P_{MiM} is path-restricted (provided P is path-restricted), proceed by induction on the choice of rendezvous, starting from the sink. To verify the base case, use the assumption that the input program is already path restricted. □

We shall reuse the core of MiM Rewrite in the forthcoming rewrites.

3.4 Additional Rewrites

This section discusses two rewrites that address proxy placement, and both extend naturally from MiM Rewrite. Interestingly, in networking, proxy placement and rendezvous selection are typically not seen as related, but the connection is clear through the lens of query optimization. This section also discusses a third rewrite that increases rendezvous flexibility by enabling off-path redirection.

(a) Original stateful server

(b) Session state at proxy

Figure 3.5: Abstract network derivation graphs for session state placement alternatives. The "loop" in 3.5a is stretched across the network to *rendezvous* in 3.5b.

3.4.1 Session Proxies

Many protocols and services maintain per-conversation session state at endpoints. However, a server may get many simultaneous connections, or multiple services might need to coexist. Either case may exhaust session state buffer space. As a result, a systems builder may prefer to offload the state to proxies (proxied server) or even to shuttle the session state back and forth with the client in each packet (stateless server). Protocols that eschew endpoint state for packet state are often termed "stateless," even though state exists in the packets. This conversion of session state is applicable in many settings (19), and is often handled manually. We next show how *Session Rewrite* can automatically and fluidly reassign session state to endhosts, packets, or proxies by simply filling in entries in a *rendezvous* relation.

Listing 3.4 shows a client-request/server-response sequence with server responses based on session state. The *Client* packages *interest* as request *message* tuples and sends these requests toward the *Server* (lines 2-3). Upon receipt, *Server* modifies *Data* in its local *session* according to the request and EDB relation *transition*, a relation capturing the protocol state machine.[3] It then returns a response *message* to the *Client* (lines 6-14). It is possible for *Client* to make followup requests by expressing more *interest* tuples, with responses dependent upon the state of *session*.

Figure 3.5a shows the abstract network derivation graph corresponding to Listing 3.4 client-

[3]Modification occurs via insertion of tuples with existing primary keys (14).

```
1  %Client: Send request message to Server.
2  message(@Client, Client, Server, Request) :-
3    interest(@Client, Server, Request).
4
5  % Server: Upon request, transition session state
6  session(@Server, Client, NewData) :-
7    message(@Server, Client, Server, Request),
8    session(@Server, Client, Data),
9    transition(@Server, Data, Request, NewData).
10
11 % ... and respond to Client.
12 message(@Server, Server, Client, NewData) :-
13   message(@Server, Client, Server, Request),
14   session(@Server, Client, NewData).
15
16 % Client: Consume response.
17 consume(@Client, Data) :-
18   message(@Client, Server, Client, Data),
19   interest(@Client, Server, Data).
20
21 % Query: What is consumed?
22 consume(@Client, Data)?
23
24 % Message forwarding used by both Server and Client.
25 message(@Next, Sender, Receiver, Payload) :-
26   message(@Current, Sender, Receiver, Payload),
27   nexthop(@Current, Sender, Next).
```

Listing 3.4: Original `SessionProg`, client-server roundtrip with session state. *session* initialization rules not shown.

request/server-response sequence with server responses based on session state. At the server, both the current *message* and *session* help to derive the next logical *message* and *session*. Also, the queried relation in this case, *consume*, is actually at the client; we are interested in

the client's status after a roundtrip communication with the server. Before discussing Session Rewrite, we first define an extension to the class of LR programs.

Definition 3.4.1. *Two IDB relations a and b are* ***linearly mutually recursive*** *(LMR) if in the rule-goal graph, there is exactly one distinct path from a to itself that visits b one time before returning to a.*

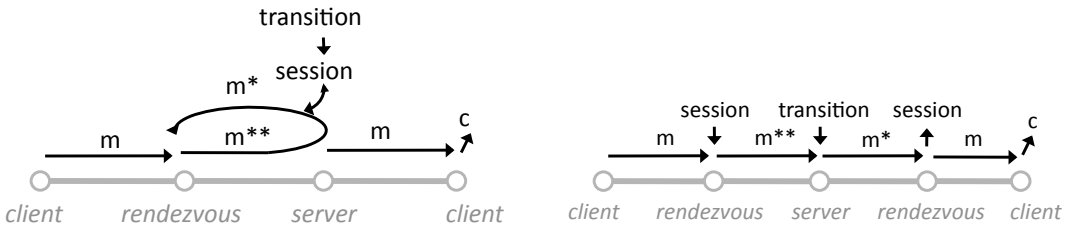
Definition 3.4.2. *A program is a* ***linearly recursive program with linear mutual recursion*** *(LR-LMR program) if it is a LR program except for some relations that are LMR.*

LR-LMR programs exhibit "linearity" equivalent to linearly recursive programs. The rule-goal graph path from a (b) to b (a) is linear (without branches), just as is the rule-goal graph path from a to a for LR relation a in a LR program.

Our primary interest in LR-LMR programs for session state is when LMR relations a and b both participate in their own LR rules, and one (say b) has LR rules for which the location specifier does not change. In such a scenario, a is analogous to messages, and b is analogous to session state. It is this pattern upon which Session Rewrite operates. For the example in Figure 3.5a, *message* and *session* map to a and b respectively.

The main idea of Session Rewrite is to use MiM Rewrite as a subprocedure, and its result is shown in Figure 3.5b. We treat *message* as if it were the queried relation, and apply MiM Rewrite to *session* as if it participated in answer rules for *message*. First, *session* generates bindings at *Server* which get pushed down into *message*'s recursion until some rendezvous r (the proxy). *message*'s recursion also arrives at r, and MiM Rewrite operates as before, returning *message_ans* to *Server*. When this occurs, answer rules may derive new *message* tuples. Because *message* and *session* are LMR, this in turn may derive new *session* tuples. The new *session* tuples generate new bindings, and are resent from *Server* back to r to seek additional joins with *message*. The net effect is that r acts as proxy for *Server*'s *session*.

Proxy selection is determined by filling in the *rendezvous* relation. Moreover, deciding among stateless, stateful and state proxy protocol variants is as straightforward as setting *rendezvous* to *Client*, *Server* or intermediate locations. The fully rewritten program after



(a) Server protocol state machine embodied as *transition*. Join keys of *transition* do not need to be shipped.

(b) Piggybacking of *session* on *message*. The same *rendezvous* is visited both on legs from and to server.

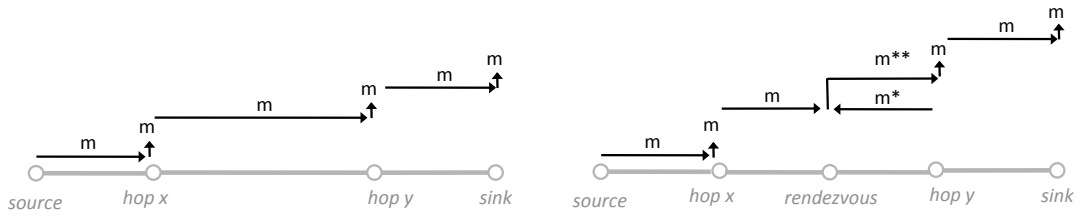
Figure 3.6: Extensions to Session Rewrite capture two common session state features.

applying path-restrictions is shown in Appendix B. We prove that our example generalizes via the following corollary to Theorem 3.3.8.

Corollary 3.4.3. *Session Rewrite is query preserving and path-restricted for LR-LMR programs.*

Proof. LMR relations are essentially LR relations with aliasing. Therefore, using MiM Rewrite as a subprocedure is query preserving for "queried relation" a by Theorem 3.3.8. This means that if either a or b participates in answer rules for actual queried relation c, c is unaffected. If they do not participate, then query preservation is trivially true. □

There are two additional features of typical session state that we also consider. First, session state changes are often based on a combination of input messages and protocol state machines. Listing 3.4 encodes the state machine as the *transition* relation. If the new session state is highly dependent upon the data in the input message (*e.g., Request* is a very selective join key in lines 6-6), then generating all possible bindings may result in a superset of *message* that is very large. To mitigate this issue, we can choose to let some join key variables remain "free", even though their values are known. This scenario is depicted in Figure 3.6a. Choosing to exclude some join key bindings does not effect Corollary 3.4.3 as long as Constraint 1 is still observed. Such an optimization is essentially the same as that proposed in (74) for traditional single-site datalog execution optimization.



(a) Original distributed routing.

(b) Hybrid source routing and distributed routing.

Figure 3.7: Routing state placement alternatives.

Second, under some circumstances, we have the opportunity to reduce messages by piggybacking. For example, consider a scenario in which *proxy* also was on the path from *server* to *client*, in addition to being on the path from *client* to *server*, as in Figure 3.6b. Then, rather than sending *message*$*$ backward from *server* to *proxy*, we could piggyback *message*$*$ onto *message* traveling from *server* to *client*, and drop off the *message*$*$ portion at *proxy*. This is possible provided the following constraint.

Constraint 4 (Proxy Revisited)**.** *The proxy must be visited on both the inbound path to and outbound path from the server.*

This implies that any subsequent *message* that goes from *client* to *server* will see the latest *session* at *proxy*, thereby preserving Corollary 3.4.3.

3.4.2 Routing Proxies

Just as servers can become overloaded with too much session state, routers can likewise exceed their capacity for holding routing state. One solution is to let packets and proxies carry the routing state instead (75). Another is to maintain routes only to a few resource-rich proxies that in turn maintain many routes (46). The *Routing Rewrite* exposes these options: it can reassign routing state to packets, proxies or some mixture of the two by filling in the *rendezvous* relation.

Specifically, we apply Routing Rewrite to distributed routing and source routing (SR) which

differ mainly in whether routing state resides in routers or packets. The prototypical message routing rule we have encountered thus far is line 7 of `BasicProg` (Listing 3.1). This resembles distributed routing akin to distance vector routing (DVR), in that nodes send *message* tuples to seek joins with *nexthop*. Conversely, SR sends *nexthop* tuples to seek joins with *message*. Routing Rewrite transforms a DVR-style program to SR, or some hybrid of DVR and SR.

Figure 3.7 illustrates an application of this procedure on `BasicProg`. Figure 3.7a explicitly draws out two hops x and y between which the optimization operates. Figure 3.7b shows the rewritten form with SR occuring between *rendezvous* and y.

Routing Rewrite is applied in the same way as Session Rewrite except that the answer rules are set to a's LR rules (such as R_1 in Listing 3.2). Consequently, the base relations generate initial bindings, and the rest proceeds as described for MiM Rewrite. For Listing 3.1 where *message* is the LR relation, this means *nexthop* generates bindings and sends these backward according to the *nexthop* relation. This relation "self-traversal" effectively mirrors what happens in networking when data about the network (such as local connectivity information) is sent on the network. The following result follows the same proof by induction pattern as Theorem 3.3.8.

Corollary 3.4.4. *Routing Rewrite is query preserving and path-restricted for LR programs.*

As with the previous rewrites, *Decision Making* can select among alternatives simply by filling in the *rendezvous* table after Routing Rewrite has been applied to the source program. To keep DVR, we set *rendezvous* to the original sink. To convert to SR, we set *rendezvous* to the source. To have some mixture of DVR and SR, we set *rendezvous* to an intermediate location. The final result of Routing Rewrite on `BasicProg` is shown in Appendix B.

3.4.3 Generalized Redirection

Thus far, our rewrites have relied solely upon on-path rendezvous and proxies. We have also extended MiM Rewrite and Session Rewrite to support redirection, a frequently used networking tool (76). Redirection opens up possibilities for alternate paths.

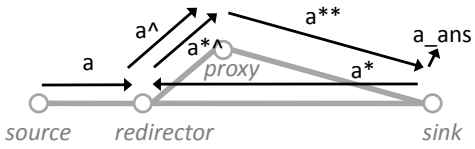


Figure 3.8: Redirector points to off-path proxy.

With the generalized redirection modification, MiM Rewrite and Session Rewrite are able to expose every network location as a potential rendezvous and proxy candidate. The modification introduces an additional three rules for each recursive relation, and a new $redirect(\overline{redirector}, \overline{rendezvous})$ relation in the EDB. Its two attribute lists represent the (on-path) redirector, and the (off-path) rendezvous. The idea is that when any message headers of LR relation a or inverted relation $a*$ encounter a matching message header $\overline{redirector}$, the message body is combined with a new message header $\overline{rendezvous}$. This leads to a and $a*$ both being redirected to the off-path proxy, as in Figure 3.8. The relations *redirect* and *rendezvous* are now both available for the optimizer to populate. Off-path proxies require additional path-restrictions for correctness.

Constraint 5 (Off-path Proxy)**.** *A path must exist from the redirector to proxy and from proxy to sink.*

As with many of the other constraints, checks on Constraint 5 require data and rule-dependent analysis. *Decision Making* choose redirectors and proxies that observe this constraint.

To summarize `netopt` thus far, given an input program: Analysis identifies recursive, base and rewrite-specific reordering candidate relations; Rewriting performs join reordering of the reordering candidate, applies path restrictions as appropriate, and, in the case of MiM Rewrite and Session Rewrite, generates additional rules for redirection. Optimization is now ready to fill in *rendezvous* and (possibly) *redirect* relations to specify the optimal execution.

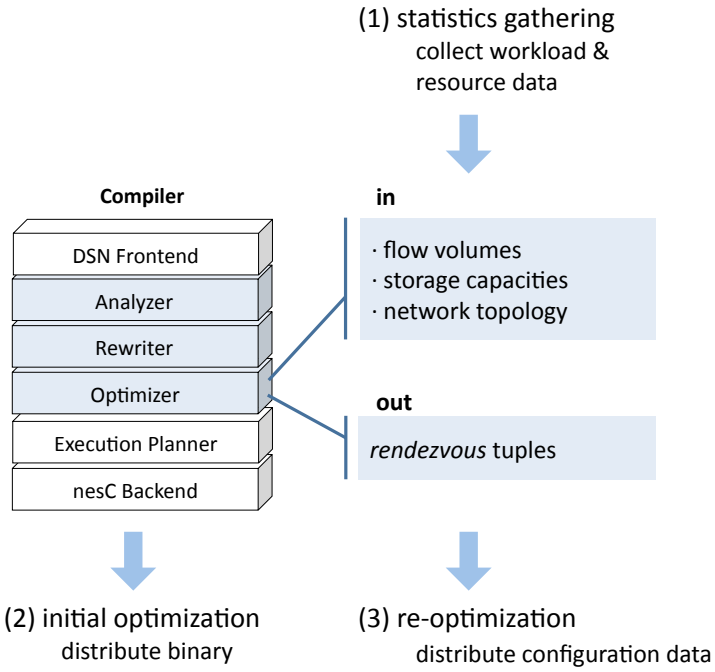


Figure 3.9: `netopt` architecture. Here it is shown embedded in the DSN compiler. We also embedded it in the Evita Raced compiler for PC-class devices.

3.5 Decision Making

The preceding sections covered the application of three rewrites to `netlog` programs to expand their possible rendezvous and proxy choices. These phases require only the source program for transformation. We now turn to *Decision Making*: searching for the optimal strategy. Figure 3.9 shows how the three steps of `netopt` are incorporated into DSN.

Inputs of *Decision Making* are network link costs and traffic profiles. Both the networking and database communities have extensively studied the problem of gathering such statistics (77; 78; 79). In the context of `netlog`, input data are all represented as relations. This information can be monitored regularly, and if sufficiently different, can trigger re-optimization.

Outputs of *Decision Making* are tuples for the relations *rendezvous* and *redirect* initial-

ized by *Rewriting*. We implemented exhaustive search algorithms for each rewrite. For MiM Rewrite, we also adapted a greedy heuristic from the networking literature (80). In principle, our rewrite-specific optimizations are replaceable by a general purpose dynamic programming optimizer, akin to those used widely by databases (81).

A benefit of the `netopt` architecture is that the analysis and rewrite to identify the optimization opportunity are distinct from the policy side of optimization. We have not focused on designing a better search algorithm for any specific scenario. Rather, we adopt an extensible framework which allows for the automated application of specific algorithms as appropriate (14; 82; 83). This permits users to drop in custom optimizers that best suite the task at hand.

3.6 Prototype Evaluation

We built a prototype `netopt` system that performs *Analysis*, *Rewriting* and *Decision Making*. The implementation uses Evita Raced, an extensible database optimizer (14), and the resulting programs run on declarative networking platforms P2 (30) and DSN. Our prototype still requires some user assistance to link together the three steps.

We evaluate the `netopt` prototype in four scenarios involving the two settings of CDNs and sensornets discussed in Section 3.1.1. We test against Emulab and Motelab testbeds (84; 85) for the CDN and sensornet settings respectively, as well as in simulation. The objective of our experiments is to measure the change in application performance over original, unoptimized programs that do not adapt to workload and resource changes. The metric to quantify performance depends upon the setting. For CDNs, we consider content access delay while for sensornets, we consider energy usage.

In both scenarios, we see optimized programs outperforming unoptimized original programs. In the CDN setting, delay is decreased by as much as two orders of magnitude. In the sensornet setting, radio operations which dominate energy consumption, are decreased by as much as one order of magnitude. In both settings, `netopt` effectively identifies and executes better strategies.

The overhead of optimization is a manageable increase in memory footprint for the installed program, as we will see in Section 3.6.5.

As with any optimizer study, the main point of our experiments is not to "invent" novel query plans (or in our case, protocol variants) that outperform well-known implementations from the literature. Rather, we wish to demonstrate that an optimizer can automatically choose variants that are well-suited to current input parameters, providing significant wins over well-known protocol variants that are not well-suited to the parameters.

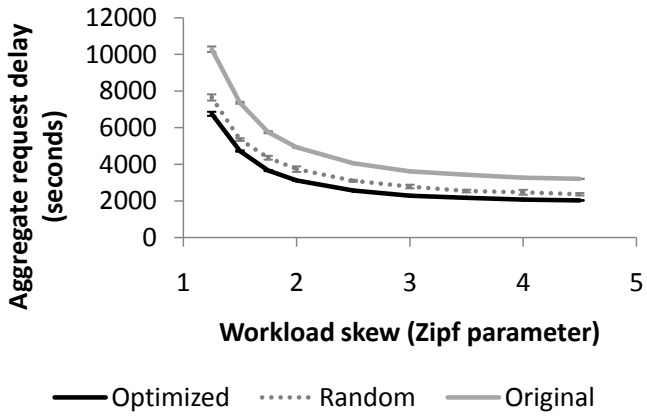
3.6.1 CDN rendezvous selection

The goal of the CDN is to decrease access time of client requests while working within storage and topology constraints. Two popular CDNs, Akamai and Limelight, differ significantly in their storage layout (66; 67). Akamai distributes content to tens of thousands of servers around the Internet, whereas Limelight maintains a few concentrated datacenters. We sought to model both during testing.

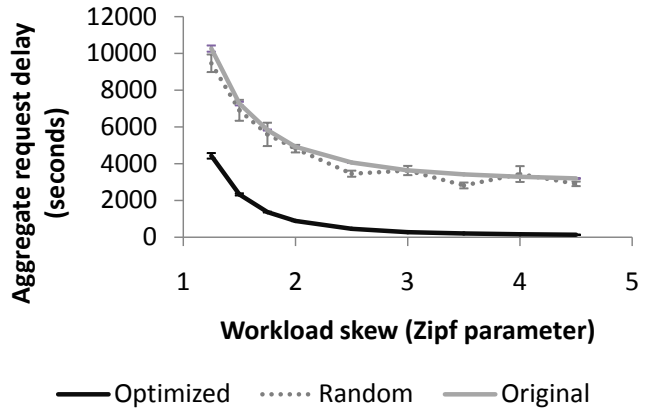
We tested by simulation and on Emulab. For simulation, we randomly generated a 200 node Internet Autonomous System (AS) topology with BRITE (86). Some nodes are selected as content producers and others as content consumers. Producers and consumers are placed at nodes of low edge degree.

Content consisted of 150 unique items. Each consumer expressed a weighted demand for each content item. This query workload was modeled as a Zipfian distribution (17). To experiment against varying workloads, we varied the skew of the Zipfian distribution.

Each node was also assigned an amount of available storage. To experiment against varying resources, we used two schemes for storage assignment, mimicking Akamai and Limelight configurations. In the first Akamai-like scheme, available storage was spread evenly among nodes. In the second Limelight-like scheme, available storage was highly concentrated at a few nodes in the network. Well-connected nodes were favored to receive available storage. The amount of aggregate storage was the same in both configurations.



(a) Concentrated storage

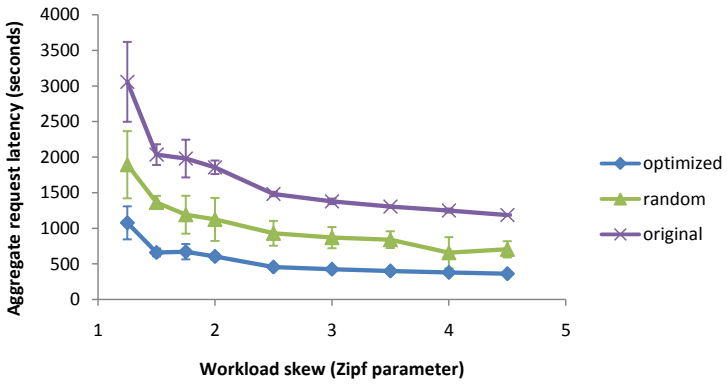


(b) Dispersed storage

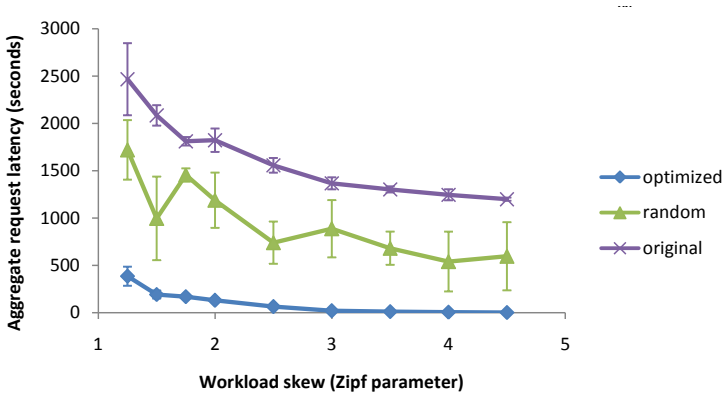
Figure 3.10: CDN rendezvous selection strategy performance under varying storage distributions and workloads in simulation.

We experimented with four CDN assignment schemes. The first, the Original scheme, consists of `BasicProg` in Listing 3.1 in which all consumer requests go directly to the content producers; available CDN storage is not utilized.

The remaining three schemes all use the rewritten `BasicProg` produced by MiM Rewrite in Listing 3.3. They differed in the *Decision Making* scheme employed, and the extent to which the schemes use workload and resource information in planning the CDN. From the standpoint of the rewritten `BasicProg`, each scheme fed in its own *rendezvous* and *redirection* relation.



(a) Concentrated storage



(b) Dispersed storage

Figure 3.11: CDN rendezvous selection strategy performance under varying storage distributions and workloads on Emulab.

The second, Random scheme, consisted of randomly assigning content items to available storage. Here, resources are fully utilized, but the workload is not considered during assignment. The third, Optimized scheme, consisted of assigning content items to available storage such that consumer requests are serviced with lowest cost. The scheme used a greedy heuristic (by order of demand weight) for this assignment adapted from the literature since the optimal assignment is known to be in NP-Complete (80). The fourth, the Exhaustive scheme, implemented the exponential version of the assignment algorithm. While the running time of Exhaustive

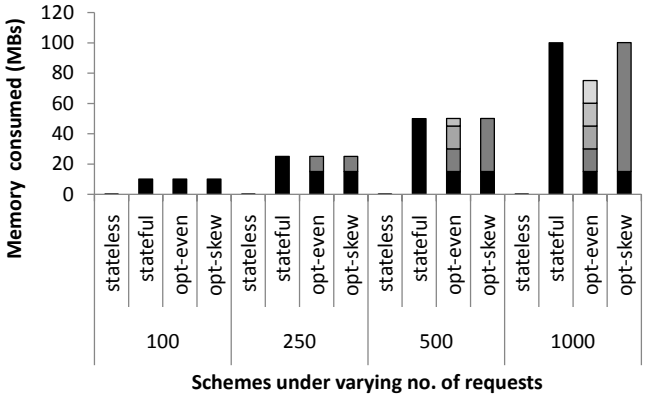
was prohibitive on our test networks, we found that in the small settings, Exhaustive made assignments that were 8-12% better than those of Optimized.

Figure 3.10 shows the results of Original, Random and Optimized schemes under varying workloads and resources. Under the Concentrated storage configuration in Figure 3.10a, Random and Optimized performed 1.3-1.4× and 1.5-1.6× better than Original respectively as the workload varies from slightly skewed to highly skewed. Under the Dispersed storage configuration in Figure 3.10b, Random and Optimized perform 0.95-1.2× and 2.3-24.4× better than Original respectively. Optimized is able to outperform Original considerably under Dispersed because it is able to assign content items to edge storage sites where there is also heavy consumer demand. Random can even underperform Original with Dispersed since poor selections can be worse than doing nothing. On Concentrated, Optimized and Random start to converge since there are relatively fewer places to choose from. In all cases, increased skew leads to lower aggregate delay because there are fewer requests that access poorly placed content in highly skewed distributions.

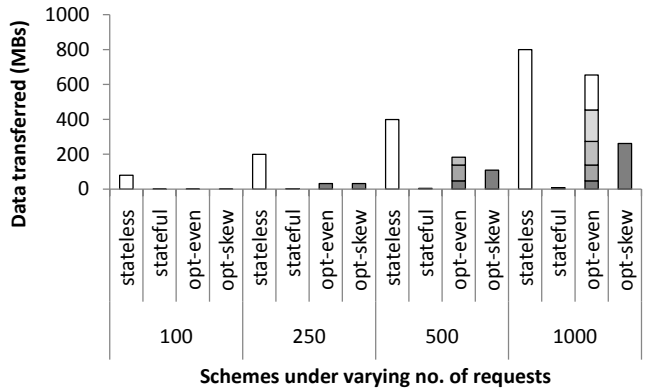
We also ran the same experiments on modest ten node Emulab networks generated by BRITE. Random and Optimized outperformed Original by 1.5-1.9× and 2.8-3.3× with Concentrated, and by 1.2-2.3× 6.4-480× with Dispersed. The trends remained the same, and are shown in Figure 3.11. These results indicate that MiM Rewrite can automatically find lower cost rendezvous points given original source program, consumer workload and network resources.

3.6.2 Server session state proxy selection

We next test the Session Rewrite. We use a five node linear Emulab network with node four making requests to node zero via nodes three, two and one. The linear network allowed us to isolate the study to the hop distance, and exclude fan-out considerations. The workload is varied from 100 to 1000 requests, with each request requiring 1Kb of session state. Two storage configurations are used, Even and Skew. In Even, each node is allotted session storage of 15Mb, which is meant to represent prime main memory. In Skew, Node One is allotted 100MB for



(a) Memory allocation



(b) Data transfer

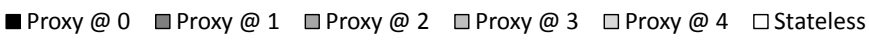


Figure 3.12: Server session state allocation strategy performance.

session state, whereas the other nodes are allotted 15MB. The Skew configuration models a scenario in which a resource rich proxy is located close to the server.

The optimization objective is to minimize the total data transfer while serving all requests. Three schemes are compared. In the first, Stateful, all session state is allocated at Node 0, regardless of whether the node storage constraint is surpassed. This corresponds to our original `SessionProg` of Listing 3.4. In the second scheme, Stateless, all session state is packaged in request and response messages. A minimal amount of storage is allocated at Node 0 to service

these stateless requests. In the third scheme, Optimized, session state is assigned to proxies so as to minimize the total data transfer. This scheme tends to use as much storage available at proxies closer to the server, Node 0, before using storage further from the server. The Optimized scheme runs the rewritten `SessionProg` shown in Listing 3.4.

Figure 3.12 shows the memory allocation and data transfer of each scheme under varying numbers of requests and storage configurations. As expected, Stateless maintains an almost negligible amount of storage usage across all nodes regardless of the number of requests, while its amount of data transfer grows very rapidly since it must package all of its request state in packets. Conversely, as seen in Figure 3.12a, storage usage under Stateful at Node 0 scales with the number of requests, well surpassing the 15MB constraint under 250 or more requests. On the other hand, the amount of data transfer with Stateful remains low even when there are many requests. Neither Stateful nor Stateless take advantage of the potential to use other nodes as proxies in the network, and therefore do not act differently when storage configurations change.

The Optimized scheme is able to take into account varying storage configurations. In Figure 3.12a, the "opt-even" and "opt-skew" labels show the resulting memory allocation on each node when the Session Rewrite optimizes against Even and Skew configurations respectively. At 100 requests when the storage limit is not yet reached, Optimized behaves just like Stateful. At higher request counts, the constraint is respected by the Optimized scheme, and storage is allocated from neighboring proxies rather than at node zero. Each segment of the stacked bars in Figure 3.12b indicates the amount of data transfer as a result of session state held at the corresponding proxy. For a given request workload, the optimized version transfers less data than Stateless, but more data than Stateful (while respecting storage constraints). This hybrid of stateless and stateful is a compromise when storage constraints are present. Furthermore, the optimizer is able take advantage of the resource-rich proxy in the Skew configuration, and transfer lower amounts of data by using Node One more when storage limits become an issue at higher request counts.

The use of proxies does come at a cost: there is a small penalty of two bytes per session

that is incurred for both storage and in each packet transfer. These two bytes are needed for the join parameters which relink a request with its session state at the proxy.

The results indicate that the rewritten `SessionProg` can have its server session state automatically assigned to proxies and packets effectively by an optimizer, lowering overall data transfer versus stateless variants. At the same time, the optimizer can automatically respect storage constraints, unlike the original stateful `SessionProg`.

3.6.3 Sensornet session state proxy selection

Next, we measure the effectiveness of Session Rewrite on sensornet programs. The optimizer attempts to minimize packets sent and received, since radio operations are often the most power-intensive activity on sensornets.

Four traditionally stateful services that have been implemented on sensornets were chosen from the literature: a network Reprogrammer, a network Debugger, an SNMP-like service, and an Interactive Shell service (87; 88; 60; 89). For each service, we estimated the state required as the service's RAM footprint as reported in the literature. These were 0.15Kb, 1Kb, 1.2Kb and 2.2Kb for Reprogrammer, Debugger, SNMP, and Interactive Shell respectively. For testing purposes, we ran placeholder programs.

These services are generally auxiliary to the main sensornet application. Therefore, the typical usage model is that it is highly desirable, though not critical, to deploy these services alongside the main application. We worked with two scenarios: the first in which the main application consumed 8Kb, and the second in which the shell consumed 5Kb, both of which we estimated from prior experience with sensornet applications. Given the mote platform we were using, this left 2Kb and 5Kb of main memory for our desired services (55).

We deployed Stateful, Stateless and Optimized programs on the Motelab testbed using DSN. The Stateful program consisted of the session state of all four services plus the main application. The Stateless program consisted of the main application, but no session state. Rather, state is transported in packets, whose data payload is a typical 20B in size (54). The

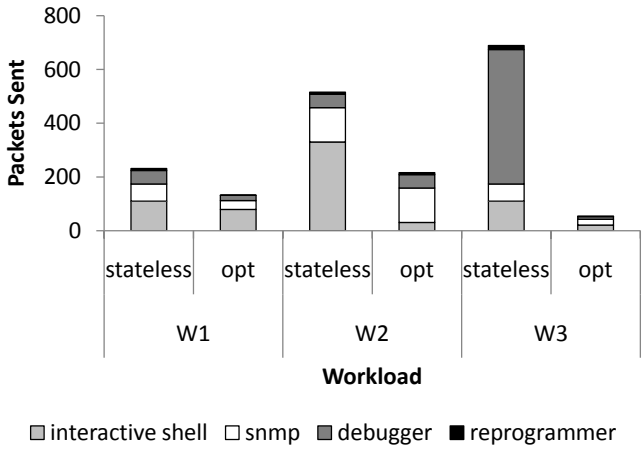


Figure 3.13: Packets sent by sensornet session state strategies.

Optimized program consisted of the main application plus a portion of each service's session state as allocated by the optimizer, with the rest pushed into packets. In each case, requests are made from a base station node across five hops to a node in the testbed that runs either Stateful, Stateless or Optimized. The workload consisted of varied distributions of calls made to each of the four services. We considered three synthetic workloads: W1, an evenly distributed workload; W2, a network monitoring workload in which SNMP and Interactive Shell were called two and three times more; and W3, a debugging workload in which Debugger and Reprogrammer were called two and ten times more.

The packets sent for the 2Kb storage limit scenario are shown in Figures 3.13. The number of packets sent for Optimal are 1.7-12.6× lower than that for Stateless, with the difference increasing as the workload becomes more skewed in W2 and W3 (Figure 3.13). Optimized allocates the most frequently called services' session state to keep on the node, thus lowering the amount of packet state necessary. Stateless, on the other hand, uses very little of the available memory, and hence is required to send more packets. It is not possible to test the packets sent for Stateful since the session state exceeds availability.

At the 5Kb storage limit, each of the four services has enough memory for its entire session state. Therefore, Stateful and Optimized perform similarly in packets sent and storage allo-

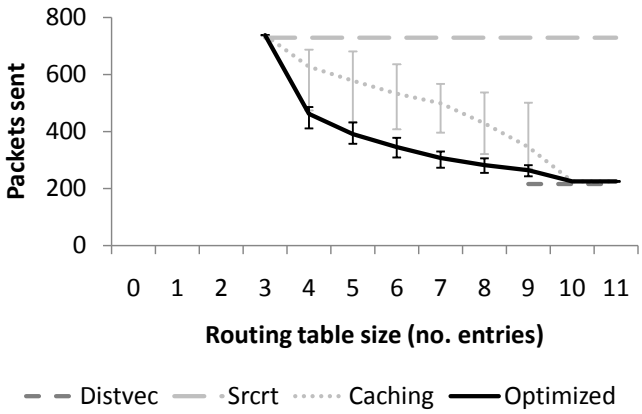


Figure 3.14: Sensornet routing state placement strategy performance.

cated. Stateless, which does not change operationally with an increased storage limit, sends the same high number of packets as before.

3.6.4 Sensornet routing state placement

Lastly, we look at routing state placement in the sensornet, and measure the ability of Routing Rewrite and optimization to choose routing state proxies. We chose a Motelab network of four hops starting from the base station. Storage is constrained such that nodes only have space for a limited number of routing entries, varying from three to eleven. Typical sensornet routing services contain four entries (54). The base station node initiated sends to nine destinations located four hops away in the network according to a Zipfian distribution.

Figures 3.14 demonstrate the results of Source Routing (SR), Distance Vector Routing (DVR), Caching, and Optimized. SR is essentially stateless, and is able to route with very few available routing entries, albeit at more packets sent. On the other hand, DVR only routes when it has enough space for all nine destinations (such that semantics were equal). Caching uses the hybrid approach of SR as the default case and residual space for DVR routing entries as requests arrive. It tends to have very high variance in packets sent due to variability of which request arrives first for caching. This variability would decrease were Caching to allow

Optimization	ROM	% Inc.	RAM	% Inc.
Rendezvous	25.2	13	3.4	63
Session	27.4	8	4.9	44
Routing	32.1	36	6.5	185

Table 3.1: Optimization Overhead in KB

for cache eviction. Optimized considers the workload such that the hotter destinations receive higher priority as DVR entries. As a result, it tends to achieve the lowest number of packets sent at all routing table sizes.

We have modeled each source route segment to correspond to one packet when in SR mode. This is a conservative assumption, and partially based on the inability of Session Rewrite and the DSN runtime to bundle multiple tuples into a single packet. Were segments to be bundled, the difference between packets sent for SR and DVR would decrease proportionally to the number of segments fitting in one packet.

3.6.5 Optimization Overhead

The overhead of optimization is primarily an increase in program rule count, resulting in larger memory footprints of optimized programs. Table 3.1 shows the optimized programs' memory usage when programs are compiled with DSN. For most cases, the increase is manageable – in the 8% to 63% relatively, and only 1.3-2.8KB in absolute terms. The outlier is Routing where the rewrite meta-application is more complex and produces many more rules than the original program. More details on the rewrite complexity can be found in Appendix B. In all cases, the programs fit comfortably on the target platform (55).

3.7 Other Application Scenarios

We chose to focus on rendezvous and proxy placement for a number of reasons: (a) they are fundamental to multiple layers of both networks and distributed systems, (b) decisions on these two fronts form key differentiators between many implementation alternatives in networks, and

(c) declarative networking bring these issues into sharper focus than they had been in other programming models. In this section we mention several other scenarios to which our results are readily applicable, due to the fundamental nature of rendezvous and proxy selection.

3.7.1 Packet filtering

Packet filtering is used to eliminate unwanted traffic based on rules about addressing, content, and volume. If a recipient node x wishes to ensure that packets are filtered on its behalf, it can either (a) receive all packets addressed to it, and filter them before processing them further (filter at receiver), (b) force all senders to evaluate packet filters (filter at sender), or (c) appoint a proxy or proxies in the network to intercept traffic between and senders and the recipient (filter at proxy). The choices amount to selecting a node or nodes where filtering rules and the messages destined for x will rendezvous; these nodes must maintain a copy of the packet filer rules for x (and must be trusted by x by some means, typically cryptographically).

3.7.2 Quality of Service

Work on providing scalable Internet quality of service has also investigated using stateless protocols and proxies in place of stateful ones (20). In these scenarios, a primary goal is to permit the large majority of core Internet routers to remain stateless while pushing quality of service state into packets and proxies at edge routers. These demonstrated that a variety of objectives such as per flow bandwidth fairness, admission control, and route pinning could all be achieved while shifting router state to packet state.

3.7.3 Publish-Subscribe

Publish-Subscribe systems have long dealt with the trade-offs of stateful vs. stateless message processing, message overhead and system implementation complexity. Historically, these systems have isolated data processing from networking, and the literature has explored the optimization space of the two (90). This work takes the view that it can be very beneficial to view

networking data just like application data, applying the same optimization techniques across both. It is even the case that some specific techniques developed for the Publish-Subscribe domain, such as specialized select-join processing methods (91), appear to map quite naturally to our rendezvous selection setting.

3.8 Summary

As sensornet workloads and resources continue to diversify, one-size-fits-all protocols are increasingly infeasible, while custom solutions require careful crafting for each environment. We investigated automatic program analysis, rewriting and optimization of network protocols along dimensions of rendezvous and proxy selection. This work naturally leads to further opportunities such as dynamic reoptimization, application to non-`netlog` programs, and new optimizations within the optimization architecture. Our study indicates that under a variety of sensornet settings, an informed optimizer can choose program executions that are much better than that of the original source program.

Chapter 4

Cross-layer Optimization

4.1 Motivation

Continuing the theme of the previous chapter, this chapter looks at network optimizations. Whereas the previous chapter explored optimizations "horizontally" across network nodes, this chapter delves "vertically" down the network stack. This chapter also turns the focus back to sensornets, where we look at the possibilities for energy savings during wireless communication.

Wireless radio communication dominates the energy consumption of many modern sensornet devices, often at a power consumption level ten to twenty times that of the processor in active mode, and one hundred times that of the processor in standby mode (55). Therefore, a wealth of energy optimization schemes have been proposed to minimize sensornet communication costs. The schemes use wide-ranging approaches at different layers of the system stack, from high-level application level data compression (92), to low-level MAC scheduling (93).

Many of these energy saving optimizations are elegant, yet few are widely deployed in practice. The most popular sensornet operating system, TinyOS, contains almost none of them ready to use. This is due to two main factors. First, it is challenging for end users to properly configure each scheme's settings to be optimal for their applications, especially when end users

may not have intimate knowledge of the mechanics of the optimization. Second, optimization designers may not have considered interactions across different optimizations. As a result, TinyOS offers users a minimalistic substrate, with the burden on the end user to assemble every optimization from scratch.

We conjecture that automating optimization decisions will spur the adoption of these many schemes because user will no longer need to worry about proper configuration settings. DSN is well-positioned to automate these optimizations because of its declarative interface. We investigate these ideas by automating three simple and well-known wireless optimizations in `wireless-netopt`, an extension of `netopt` for wireless networks. Each optimization occurs at a different level of the network stack. As we will see shortly, each is capable of energy savings from a few percent up to 3×. Hence, each of the optimizations is immediately useful for building longer-lasting sensornets.

Of course, it is possible to apply these three optimizations one after the other in sequence as *independent optimizations*. But, it is often the case that the combination of locally optimal solutions does not necessarily lead to global optimum. The alternative is to consider *joint optimization*, where parameters for all optimizations are considered simultaneously.

We show two main results for joint optimizations composed in `wireless-netopt`. First, interfacing individual optimizations for joint optimization is quite feasible, and is aided by the ease of `netlog` program analysis and rewriting. Second, jointly solving all three optimizations yields significantly greater savings than independent sequential application. We use a combination of simulation and sensornet testbed experimentation to compare unoptimized, independently optimized, and jointly optimized solutions. In simulation, optimal solutions use 17× less energy than unoptimized solutions. On the sensornet testbed, optimal solutions beat unoptimized solutions by over 2×.

In addition, this chapter charts the design and implementation of these optimizations in a declarative sensornet programming system, `wireless-netopt`. As a main result of this exercise, we demonstrate that the declarative programming model's simplicity makes program analysis

and program transformation very easy. It is not difficult to incorporate new optimizations even when they are specific to particular networking domains, such as sensornets.

4.1.1 Organization

This chapter is organized as follows. Section 4.1.2 introduces the three specific network optimizations that we seek to study in this work. Section 4.2 formalizes these optimizations, presents algorithms for solving them, and shows how `wireless-netopt` can automatically analyze and transform source programs to benefit from the optimizations. Section 4.3 presents the implementation and evaluation of `wireless-netopt`. Section 4.4 summarizes this chapter's results.

4.1.2 Optimizing Energy Consumption in the Networking Stack

We look at three specific and separate optimizations in the networking stack: (1) setting receivers' polling intervals, (2) choosing whether to broadcast or unicast packets, and (3) deciding whether to redirect packets through a one-hop intermediary. Figure 4.1 shows the typical layers in the networking stack where these decisions are made. For example, typically the Broadcast vs. Unicast decision is made by the Network layer and carried out by the MAC layer.

Furthermore, we focus on a wireless local area mesh, where each node is connected directly to others in the local area. For a fair number of applications, the entire application is one mesh (94; 95; 96). For other applications, multi-hop networking is expected, with multiple local areas stitched together to form the multi-hop network. Although our discussion focuses on a mesh, our results apply to multi-hop settings as well. As in the previous chapter, we seek to take advantage of heterogeneity, either in the workload distribution or in the resource distribution.

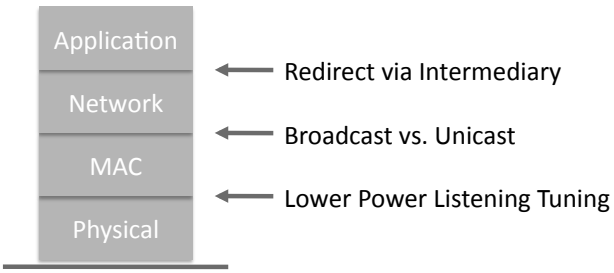


Figure 4.1: The networking stack and points at which three optimizations are typically made.

Low Power Listening

Radio operations dominate energy consumption on modern wireless sensornet platforms. Radio operations can be broken down into five modes: transmit, receive, carrier sense, idle listening and sleep. The first four modes draw approximately the same amount of power, whereas sleep mode is often several orders of magnitude less power intensive (55). For a large class of sensornet applications, data rates are low enough to warrant switching off the radio when there are neither ongoing transmissions nor receptions (97). Therefore time spent in *idle listening* – that is, in active radio receive mode when not actually receiving – is particularly deleterious to network lifetime because of lost sleeping opportunity.

Many MAC protocols have been proposed to maximize sleep time and minimize idle listening (93). Among these, Low Power Listening (LPL) is the default MAC in many sensornet systems (54; 98), and is illustrated in Figure 4.2. It is a variant of a carrier sense protocol and works as follows: a receiver and sender agree on some period t_p. The sender can send a packet at any time, provided that it prefaces every packet with a preamble of length t_p. The receiver is guaranteed to receive the packet as long as it wakes up every t_p to check the channel. LPL presents a direct trade-off between the sender cost vs. receiver cost; increasing t_p lowers the idle listening incurred by receivers while raising the preamble burden for senders.

Given application-level data rates, finding the optimal LPL period is straightforward (99). It is possible to make this calculation either at coarse granularity, with one t_p for the entire

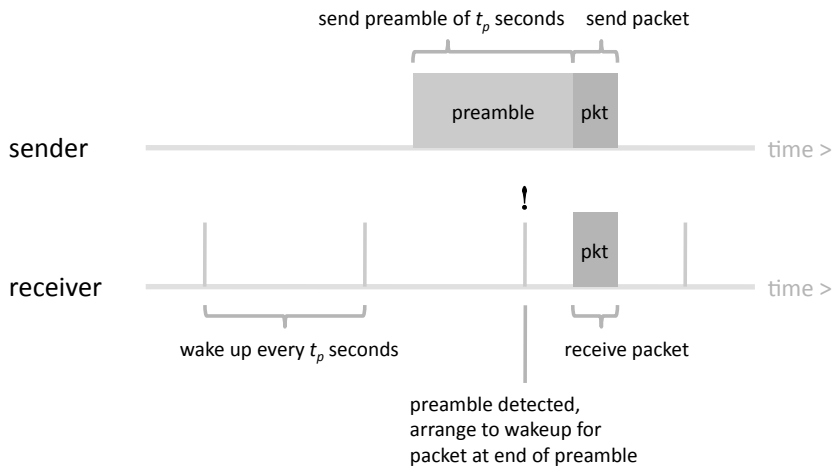


Figure 4.2: Low power listening MAC protocol. Both sender and receiver agree on polling interval t_p a priori.

network, or at fine granularity, with one t_p per node. Yet, even with this modest degree of deployment-specific input required, this exercise is rarely carried out in practice at either the coarse or the fine grain. Automatic optimization should be able to deliver the benefits of this mechanical tuning task without overburdening the user.

Broadcast or Unicast

Our second optimization considers how to ship data: either by unicast or by broadcast. A sender can decide to either to communicate one-to-one directly with each desired receiver (*unicast* communication), or to communicate one-to-all with every node (*broadcast* communication). For unicast, the send and receive costs are proportional to the number of intended destinations, which could be lower than the size of the entire network. For broadcast, the send cost is the cost to send a single packet. However, the receive cost is proportional to the number of nodes in the local area, since every node will receive the packet, even those that are not an intended destination.

The trade-off between broadcast and unicast can be characterized by the degree of *payload*

sharing, which can range from one to the size of the local area. Lower payload sharing indicates that fewer receivers are interested in a given payload and unicast may be a better option, whereas higher payload sharing indicates that more receivers are interested in a given payload and broadcast may be a better option.

The decision of when to use unicast or broadcast can dramatically impact the design of a system, wireless or wired. For instance, much research in the field of replicated systems concentrates on when to use each mode of communication to maximize throughput and minimize response times, and how to architect entire protocols around that decision (100; 101). The best choice depends not only upon the number of intended destinations (which is application-dependent) but also upon the relative costs of transmission and reception (which is dependent on the radio technology used). It is difficult and unusual for programmers to consider these factors jointly, so this is a compelling target for automatic optimization.

Redirection via Intermediary

Our third optimization looks at how senders might benefit from redirecting packets to an intermediary, rather than sending directly to receivers. Chapter 3 dealt extensively with rendezvous selection across a network. Here, we only consider redirection to an intermediary restricted to the local area. Also unlike Chapter 3, the intermediary is not "on-path" – that is, it is not on the original route between sender and receiver.

It turns out that even this limited case of redirection presents very useful opportunities for optimization, especially in cases where there are high degrees of node heterogeneity. First, just as in the broadcast case, the sender has the opportunity to reduce transmission costs by taking advantage of payload sharing when redirecting to the intermediary. For example, suppose a sender needs to transmit the same payload to several receivers. Rather than directly sending the payload multiple times, the sender can pass the payload once to the intermediary, and let the intermediary take the on responsibility of forking the packet for multiple sends. This makes the most sense when payload sharing is high, and the intermediary is energy-rich.

Second, both sender and receiver can take advantage of discrepancies in the intermediary's

LPL settings. A well-provisioned intermediary may be capable of frequent idle listening; the sender can arrange to transmit to the intermediary with a very short preamble. Additionally, the well-provisioned intermediary may be able to send very long preambles; the receiver can wake up at very long intervals to receive messages from the intermediary. In this way, both sender and receiver transmit and receive at very low cost with the assistance of the intermediary. Note that this second potential benefit of off-path intermediary emerges naturally if intermediary choices are considered jointly with optimal LPL settings, but is otherwise only possible opportunistically.

4.2 Optimizations

This section formalizes the three optimizations of `wireless-netopt`.

- `Redirection-via-Intermediary`
- `Broadcast-or-Unicast`
- `LPL-Tuner`

Each optimization opens with an overview of the optimization problem, and then looks at how program rewriting transforms the input program to the optimal output program.

The optimizations ultimately seek to minimize the cost of communication. Specifically, we seek to minimize the maximum energy expenditure at any one node as a percentage of each node's energy level. In other words, we aim to limit the worst case node energy drain. This objective is especially sensible when each node's individual functionality is important to the overall network's operation. Though we could have also looked at cumulative node energy costs and other metrics, these do not fundamentally alter the application of `wireless-netopt`.

We continue by building on our use of `netlog`, and introduce the concept of a *rule-sender*, a pair of one rule and one sending node. The rule-sender is the granularity at which `wireless-netopt` performs optimization. The rule must be a *distributed rule*, meaning that the location specifier for the head must differ from the location specifier for the body. The rule-sender unit

is a good granularity at which to make decisions. A rule by definition groups a batch of logically related tuples. Were we to smooth optimization across rules, we might lose naturally occurring intra-rule correlations. Similarly, were we to smooth optimization across nodes, visibility of high inter-node variability would be lost.

Associated with each rule-sender are two statistics. The first is a map of receivers to data rates. A flow is defined as a sender, receiver and data rate. Therefore, a rule-sender with its map of receivers and data rates defines one or more flows. The second statistic for each rule-sender is the degree of payload sharing (PS) for any payload produced by this rule-sender. Intuitively, a high PS indicates that more receivers can potentially share the same broadcasted tuple.

4.2.1 Redirection-via-Intermediary

We tackle `Redirection-via-Intermediary` in two stages. First, we focus on the case of optimizing a single rule-sender. This can be solved in time linear to the size of the local area network. Second, we look at the general problem of optimizing multiple rule-senders. This problem is in NP, and therefore we suggest a greedy algorithm that runs in time cubic to the size of the local area network. This is tolerable since we only anticipate local area sizes in the range of a dozen nodes or fewer.

Optimization Problem

Given a single rule-sender, the problem is to select the best intermediary node i such that the maximum percentage energy drain of any node is minimized. We restrict the intermediary to one hop; we do not consider chains of intermediaries and we are not trying to solve the general routing problem. In the local area mesh, there are significantly diminishing returns for considering routes greater than one hop unless pairwise connectivity is systematically asymmetric, which is unusual.

Finding the best intermediary can be accomplished by evaluating each node $n \in N$ as the

```
planEval(ruleSender,intermed)
  totalCost = 0
  totalVolume = 0
  mac = getMacModel()
  totalCost += mac.send(ruleSender.sender,intermed,totalVolume/ruleSender.ANR)
  for (receiver,volume) in ruleSender.receiverVolumes
    totalCost += mac.send(intermed,receiver,volume)
    totalVolume += volume
  return totalVolume/totalCost

mac.send(from,to,volume) //... to be defined
```

Listing 4.1: Plan Evaluation

intermediary, where N is the set of nodes in the network. We perform this evaluation with the function $planEval(ruleSender, n)$, shown in Listing 4.1. The weight assigned to the plan is the total volume shipped divided by the total ship cost.

So far, we have not yet specified how an over-the-air *mac.send*() is evaluated. This evaluation can differ, depending on whether `Redirection-via-Intermediary` is to be considered independently from, or jointly with other optimizations. When the `Redirection-via-Intermediary` is considered independently from other optimizations, a simple model is used where transmission and reception costs are inversely proportional to node energy levels. Section 4.2.4 discusses the evaluation implications when `Redirection-via-Intermediary` is considered jointly with other optimizations.

For multiple flows, finding the best intermediaries maps to the NP-hard multi-commodity flow problem (102). Therefore, we employ an iterative greedy algorithm shown in Figure 4.2. On every major iteration, the *ruleSender* with the best *planEval* score is assigned to the intermediary that led to that best score. Any node that incurred transmissions or reception costs due to that *ruleSender* has those costs deducted permanently from its energy budget. All unassigned rule-sender have their scores reevaluated with respect to the updated node energy levels before another major iteration to select the next best *ruleSender*. In Section 4.3, we show that our iterative algorithm performs well in practice.

```
solve(ruleSenders,nodes)
  for i in 0..len(ruleSenders)
    allPlans = [] // pairs of (score,ruleSender) sorted by score
    for ruleSender in ruleSenders
      for intermed in nodes
        allPlans.add((planEval(ruleSender,intermed),ruleSender))
    (bestScore,bestRuleSender) =allPlans.top()
    assignPlan(bestRuleSender) // deducts node energy costs as well
    ruleSenders.remove(bestRuleSender)
```

Listing 4.2: Multi-flow Iterative Solver

Program Analysis and Rewriting

Recall that in general, a distributed rule can take the following form.

```
head(@Dst,...) :- body_1(@Src,...), ..., body_N(@Src,...).
```

The following distributed rule, `DR`, serves as our running example instance of the general distributed rule.

```
recv(@Dst,Fid,Uid) :- send(@Src,Dst,Fid,Uid).
```

The intent of `DR` is that *Src* generates tuples for *Dst* with some flow id *Fid* and some unique id *Uid*. It is not consequential that the body only contains one predicate. We can simplify all distributed rules to have one body predicate as a pre-processing stage if necessary. `Redirection-via-Intermediary`, makes the following transformation on `DR`.

```
% Original execution: no intermediary
recv(@Dst,Fid,Uid) :- send(@Src,Dst,Fid,Uid), xl_noproxy_RN(@Src).

% Rewrite (part 1): source redirects to intermediary
xl_intermed_RN(@Xl_prx,Dst,Fid,Uid) :- send(@Src,Dst,Fid,Uid),
    xl_proxy_RN(@Src,Xl_prx).

% Rewrite (part 2): intermediary sends to destination
recv(@Xl_dst,Fid,Uid) :- xl_intermed_RN(@Xl_prx,Dst,Fid,Uid).
```

Three rules are emitted. The first rule retains the original execution where *Src* directly sends to *Dst*. However, note that the rule does not execute unless an appropriate entry in *xl_noproxy_RN* exists. The second and third rule realize the alternative execution where *Src* sends indirectly to *Dst* via *Xl_prx* using the relation *xl_intermed_RN*. As with the rewrites in Chapter 3, the optimizer installs its chosen intermediaries by filling in entries in configuration tables. In this case, the configuration tables *xl_proxy_RN* and *xl_noproxy_RN* simply indicate which rule-senders are getting redirected via an intermediary, and the identity of that intermediary.

During actual `Redirection-via-Intermediary` optimization, the predicate suffix *_RN* is replaced with the system-assigned rule number for the original distributed rule `DR`. This allows us to differentiate the rewriting and optimization of multiple distributed rules.

The rewrite shown so far does not perform payload aggregation for shared payloads. We will show this part of the rewrite in conjunction with `Broadcast-or-Unicast` next.

4.2.2 Broadcast-or-Unicast

Optimization Problem

For each rule-sender, the problem of deciding whether to broadcast or unicast is nominally straightforward. Given the number of receivers, network size, send cost, and receive cost, the following comparison can be made:

$$\begin{aligned} UnicastCost &= (NumReceivers)(SendCost + RecvCost) \\ BroadcastCost &= SendCost + (NetSize)(RecvCost) \end{aligned}$$

NumReceivers is dependent upon the degree of payload sharing, and can be measured from workload observation. If a sender sends the same payload to many receivers, then *NumReceivers* is high. Otherwise, it is low.

The modeling of *SendCost* and *ReceiveCost* depend on whether `Broadcast-or-Unicast` is optimized independently from or jointly with other optimizations. When `Broadcast-or-Unicast`

is optimized independently from other optimizations, *SendCost* and *ReceiveCost* are modeled after physical radio properties, and on relative node energy levels only. A common simplification is to assume that sending an receiving are of equivalent cost, in which case *SendCost* and *ReceiveCost* are both inversely proportional to a node's energy level. Section 4.2.4 discusses the important implications when `Broadcast-or-Unicast` is considered jointly with other optimizations.

Program Analysis and Rewrite

The analysis phase of this optimization is not complicated, and simply looks for distributed rules. In particular, if `Broadcast-or-Unicast` runs after `Redirection-via-Intermediary`, any distributed rules generated by `Redirection-via-Intermediary` are candidate rules.

The main idea of the rewrite is to generate a set of rules that decide whether to use a unicast or a broadcast version of the rule for a particular *payload* produced by the rule. Payloads consist of tuples in the rule head with the location variable excluded. For example, given the head predicate $recv(@Dst, Fid, Uid)$, the payloads for *recv* are tuples with attributes Fid and Uid. The following rewrite retains options for both the original unicast execution and the broadcast execution, and lets the optimizer choose between the two.

```
% Original Execution: Unicast
recv(@Dst,...) :- send(@Src,...), xl_ucast_RN(@Src).

% Rewrite (part 1): Aggregate payloads
recv_payload(@Src,SET<Dst>,...) :- send(@Src,...), xl_bcast_RN(@Src).

% Rewrite (part 2): Broadcast each payload once
recv_broadcast(@*,DstSet,...) :- recv_payload(@Src,DstSet,...).

% Rewrite (part 3): Receive payload only if at intended destination.
recv(@Snk,...) :- recv_broadcast(@Snk,DstSet,...), @Snk in DstSet.
```

Part 1 of the rewritten execution is a rule that buffers each unique payload along with

the destination set that indicates where the payload is to be sent. This is done by grouping into distinct payloads and aggregating along the location variable *i.e.,* a group-by aggregation operation.

The *SET* aggregation operator forms the set of all destinations for each unique payload. It is a straightforward user-defined aggregate. For example, assume the following rule runs at *node*0.

```
recv(@Dst,Fid,Uid) :- send(@Src,Dst,Fid,Uid).
```

If the result set of *recv* is the tuple set:

```
recv(@node1,flowA,uidB).
recv(@node2,flowA,uidB).
recv(@node1,flowA,uidC).
```

then the tuples generated by part 2 are:

```
recv_payload(@node0, {node1,node2}, flowA, uidB).
recv_payload(@node0, {node1}, flowA, uidC).
```

where the second field contains the set of intended destination for each unique payload.

Part 2 of the rewrite performs the actual broadcast of *recv_payload*. Since every node in the local area will receive this tuple, Part 3 has each node look in the set *DstSet* to check whether or not it is an intended destination. Those that are in the set convert the received tuples into *recv* tuples. Once again, the exact choice of broadcast vs. unicast is decided by entries in the *xl_bcast_RN* and *xl_ucast_RN* tables. These are populated by `Broadcast-or-Unicast`.

4.2.3 LPL-Tuner

We now refine our notions of message *SendCost* and *ReceiveCost* introduced earlier. Fundamentally, there is some necessary amount of coordination required to communicate, and an optimization decision is to decide how to split this coordination cost between sender and receiver. Previous work has formulated this problem before for specific MAC implementa-

tions (99). We tackle the optimization of parameters for a particular MAC, LPL, used in many sensornet deployments (103).

Optimization Problem

Previous work derived optimal LPL settings based on the data rate from sender to receiver (99). Briefly, if the sender is often sending data, the receiver should wake up frequently to check the channel. Alternatively, if the sender is rarely sending data, the receiver should be allowed to sleep for long intervals between checks. We make two extensions to the analysis (99). First, we are interested in arbitrary communication patterns, rather than fixed all-to-all communication used in (99). Second, we are interested in node heterogeneity, and specifically in cases where some nodes have orders of magnitude more energy than others. Our goal is to minimize node energy drain as a percentage of energy budget.

At the moment, consider the case of a single sender and receiver, with data rate r_{data} (Recall that a rule-sender, may have more than one receiver, each with its own data rate). We seek to minimize the combined energy expenditure of sender and receiver as a percentage of energy level (capacity).

$$E_{total} \quad = \quad E_{sender}/C_{sender} + E_{receiver}/C_{receiver} \tag{4.1}$$

A summary of the variables used in our analysis are provided in Table 4.1. The sender and receiver energy expenditures are the sum of energy spent in carrier sense, transmission, polling, reception and sleep modes. Energy expenditure in each mode can further be broken down into the power consumed and time spent in the respective mode.

Variable	Description
t_p	polling interval
r_{data}	data rate
$E_{sender}, E_{receiver}$	sender/receiver energy expenditure
$C_{sender}, C_{receiver}$	sender/receiver energy level
t_{cs}	time in carrier sense mode
t_{tx}	time transmission mode
t_{poll}	time in channel poll mode
t_{rx}	time in reception mode
t_{sleep}	time in sleep mode
Constant	Description
P_{cs}	power to perform carrier sense
P_{tx}	power to perform transmission
P_{poll}	power to perform channel poll
P_{rx}	power to perform reception
P_{sleep}	power to sleep
t_{csl}	time to sense carrier
t_{pl}	time to poll channel
t_{pkt}	time to send/receive a packet

Table 4.1: Low Power Listening parameters

$$E_{sender} = E_{cs} + E_{tx} + E_{sleep} \tag{4.2}$$

$$= P_{cs}t_{cs} + P_{tx}t_{tx} + P_{sleep}t_{sender\_sleep} \tag{4.3}$$

$$E_{receiver} = E_{poll} + E_{rx} + E_{sleep} \tag{4.4}$$

$$= P_{poll}t_{poll} + P_{rx}t_{rx} + P_{sleep}t_{receiver\_sleep} \tag{4.5}$$

The power consumed in each mode is a function of the radio, and can be obtained from manufacturers' datasheets (104). The times spent in each mode is dependent upon the radio, the data rate r_{data} and polling interval t_{data}.

$$t_{cs} = t_{csl} r_{data} \quad (4.6)$$

$$t_{tx} = (t_p + t_{pkt}) r_{data} \quad (4.7)$$

$$t_{poll} = t_{pl}/t_p \quad (4.8)$$

$$t_{rx} = (2.5 t_{pkt}) r_{data} \quad (4.9)$$

$$t_{sender\_sleep} = 1 - t_{cs} - t_{tx} \quad (4.10)$$

$$t_{receiver\_sleep} = 1 - t_{poll} - t_{rx} \quad (4.11)$$

We briefly describe Equations (4.6)-(4.11) above, each of which defines the time spent in a particular radio mode with respect to a unit time interval (*i.e.,* 1 second). Equations (4.6)-(4.9) were originally formulated in (99). Equations (4.10)-(4.11) are variations of formulas found in (99). Equation (4.6) defines the time in carrier sense mode as a single carrier sense operation's time scaled by the data rate. Equation (4.7) defines the time in transmission mode to be the time needed to transmit the preamble plus the time to send a packet scaled by the data rate. Equation (4.8) defines the time spent in polling mode as the time to perform one poll inversely scaled by the polling interval. Equation (4.9) defines the time spent in reception mode as the time to receive 2.5 packets scaled by the data rate. The 2.5 packets come from our usage of the BMAC+ protocol, an improved version of the standard LPL protocol (103). In the BMAC+ protocol, the first *preamble packet* is received when a polling check on the channel succeeds. Its payload contains a counter that indicates when the receiver should wake up to receive the actual data packet. On average, the receiver will wake up and need to wait half the length of a packet before receiving the start of a preamble packet. The second packet is the actual *data packet*. Equation (4.10) and Equation (4.11) define the time spent sleeping as the time not spent in other modes. From these equations, it is clear that a longer polling interval both (1) increases transmission time and (2) decreases polling time.

To find the optimal polling interval and minimal energy expenditures, we differentiate Equation (4.1) with respect to t_p, set the result to zero, and solve for t_p. The result is as follows, where the parameters $K_1 - K_7$ are arithmetic combinations of the constants in Table 4.1.

$$t_p* = K_1\sqrt{\frac{C_{sender}}{C_{receiver}r_{data}}} \quad (4.12)$$

$$E_{sender}* = K_2 + K_3 r_{data} + K_4\sqrt{\frac{C_{sender}r_{data}}{C_{receiver}}} \quad (4.13)$$

$$E_{receiver}* = K_2 + K_5 r_{data} + K_4\sqrt{\frac{C_{receiver}r_{data}}{C_{sender}}} \quad (4.14)$$

Several properties of this relationship are worth mentioning. First, higher data rates cause shorter polling intervals. Second, increases in sender energy relative to receiver energy cause longer polling intervals, more sender energy expenditure, and lower receiver energy expenditure.

This analysis can be similarly extended to the case of multiple receivers of varying energy capacities. The objective is to minimize the total energy which is now defined with respect to the sender and receivers 1..n.

$$E_{total} = E_{sender}/C_{sender} + E_{r,1}/C_{r,1} + \ldots + E_{r,n}/C_{r,n} \quad (4.15)$$

The optimal polling interval and minimum receiver energy expenditures are revised as follows.

$$t_p* = K_6\frac{Combo}{Sequ}\sqrt{\frac{C_{sender}}{r_{data}}} \quad (4.16)$$

$$E_{sender}* = K_2 + K_3 r_{data} + K_7\frac{Combo}{Sequ}\sqrt{C_{sender}r_{data}} \quad (4.17)$$

$$E_{r,i}* = K_2 + K_5 r_{data} + K_8\frac{Sequ}{Combo}\sqrt{\frac{r_{data}}{C_{sender}}} \quad (4.18)$$

$$Combo = \sqrt{K_9\sum_{1\le i\le n} C_{r,1}\ldots C_{r,i-1}\ldots C_{r,i+1}\ldots C_{r,n}} \quad (4.19)$$

$$Sequ = \prod_{1\le i\le n} C_{r,i} \quad (4.20)$$

In the formulation above, r_{data} is the average data rate for all receivers 1..n. In this work, we use the Tmote platform and its CC2420 radio (104). The radio-dependent constants for this case are shown in Table 4.2.

Constant	Value
K_1	0.0243
K_2	0.003
K_3	0.180
K_4	1.27
K_5	0.108
K_6	0.000379
K_7	0.0198
K_8	81.1
K_9	4099

Table 4.2: Optimal polling interval and minimum energy expenditure parameters derived from CC2420 radio-specific constants.

Program Transformation Mechanics

`wireless-netopt` assigns the optimal polling interval t_p* to the user program with the help of *builtins*, introduced in Section 2.2. We use the *builtin_preambleLen* and *builtin_pollingInt* builtins as controls for adjusting the send and receive cost. For example, sender node *node*0 and receiver node *node*1 can set these with facts as follows.

```
builtin_preambleLen(@node0, preambleLen).
builtin_pollingInt(@node1, pollingInt).
```

These facts set the preamble length and polling interval statically per node. However, this does not meet our initial requirement of setting the preamble per rule-sender. This requirement can be met by setting the preamble length dynamically on a per rule basis. This is accomplished with a rule transformation, which we demonstrate with the example distributed rule `DR`.

```
builtin_preamble(@Src, Xl_preamble ) :-
  send(@Src, Dst, Fid, Uid),
  xl_preamble_RN(@Src, Xl_preamble).
recv(@Dst, Fid, Uid) :-
  send(@Src, Dst, Fid, Uid),
  builtin_preamble(@Src, Xl_preamble).
```

In the example above, the first rule sets *builtin_preamble* based on the entry in the *xl_preamble_RN*, which is a configuration table set on a per rule-sender basis. After the setting is complete, the second rule sends the actual payload. This transformation is equally applicable to any distributed rule, including the output of `Redirection-via-Intermediary` and `Broadcast-or-Unicast`.

`LPL-Tuner` illustrates a different approach to optimization than the previous two optimization encountered so far. Even though the exact mechanics of the MAC external module are opaque, it can still be modeled analytically. This is enough for `wireless-netopt` to gather optimization inputs and make optimization decisions off of it via a relation-like builtin interface. Combined with `Redirection-via-Intermediary` and `Broadcast-or-Unicast`, `LPL-Tuner` potentially lowers the total processing cost of a given distributed rule by choosing the best application-dependent LPL parameters.

4.2.4 Joint Optimization

Thus far we have discussed three optimizations and how they can each be independently applied in sequence. It is possible to further improve the result by jointly solving all three optimization problems.

First, consider the combination of `Redirection-via-Intermediary` and `LPL-Tuner`. When an energy-rich intermediary is in the presence of a sender and receiver, the independent optimization of `Redirection-via-Intermediary` will not choose to use the intermediary if payload aggregation alone offers no gain for switching to an intermediary. However, in conjunction with `LPL-Tuner`, the intermediary can offer very substantial gains by using a very short preamble with the sender, while using a very long preamble with the receiver.

Second, consider the combination of `Broadcast-or-Unicast` and `LPL-Tuner`. `Broadcast-or-Unicast`'s decision of whether to send broadcast messages or unicast messages effects the set of receivers. This impacts decisions made by `LPL-Tuner` in setting the best preamble length. Likewise, the preamble length impacts whether broadcast or unicast is more efficient; a long preamble favors broadcast with fewer transmissions.

Table Name	Description
$xl\_noproxy\_RN(@Sender)$	This rule-sender does not use an intermediary.
$xl\_proxy\_RN(@Sender, Xl\_prx)$	This rule-sender uses the intermediary Xl_prx.
$xl\_ucast\_RN(@Sender)$	This rule-sender uses unicast.
$xl\_bcast\_RN(@Sender)$	This rule-sender uses broadcast.
$xl\_preamble\_RN(@Sender, Xl\_preamble)$	This rule-sender uses preamble $Xl\_preamble$.

Table 4.3: Configuration tables that are populated by the optimizer, and the implications of an entry in the table.

It is also possible for the combination of `Redirection-via-Intermediary` and `Broadcast-or-Unicast` to benefit from joint optimization. However, we project the gains to be quite minor, since an energy-rich intermediary picked by `Redirection-via-Intermediary` will unlikely need to resort to broadcasting, which is meant to save the sender energy. Therefore we do not consider it further.

To interface the optimization algorithms to one another, we make the following modifications. The method *mac.send* in Listing 4.1 for `Redirection-via-Intermediary` links to `Broadcast-or-Unicast`. The *SendCost* and *ReceiveCost* variables in Equation (4.1) link to `LPL-Tuner`. Ultimately, both independent optimization and joint optimization set the following configuration tables summarized in Table 4.3.

As mentioned on several occasions in this section, the actual rewriting of each optimization operates on distributed rules, including those emitted by other optimizations.

4.3 Implementation and Evaluation

We implemented each of the three optimizations discussed in Section 4.2 into the DSN compiler. In addition to the source program, the compiler takes as input rule-sender data. Recall that an rule-sender specifies a rule, a sender, a mapping of receivers to data rates, and an PS. The rule-sender data can be collected much like any other network statistic. Techniques for doing so were discussed in Section 3.5. We assume that such rule-sender data is available.

Our evaluation consists of both simulation and sensornet testbed evaluation. In simulation,

we test many different types of network and workload configurations at larger scale. On the testbed, we validate our simulation results, as well as directly measure the time spent in each of the various radio operational modes. As a summary of our experiments, energy savings can be as much 17× in larger-scale simulated networks, and are over 2× in smaller scale testbed networks. The greatest savings are possible when networks are heterogeneous, and when joint optimizations are employed. Furthermore, our optimizer is able to model actual energy consumption very accurately, with over 0.9 correlation between optimizer's expected energy consumption and actual energy consumption.

4.3.1 Simulation

All simulations used a ten node network with five rule-senders. Ten nodes is a reasonable neighborhood size for the local area (105). rule-sender traces were generated by sampling uniformly from data rates of 10, 1, .1 and .01 packets per second. Senders and receivers were chosen uniformly from all nodes. The following parameters were investigated in simulation.

- *Energy Heterogeneity.* We experimented with both homogeneous and heterogeneous node energy distributions. In the homogeneous setting, every node started with the same base energy level. In the heterogeneous setting, nodes modeled a tiered environment, with one node having 100× the base energy level, and two nodes having 10× the base energy level.

- *Number of Receivers.* Each rule-sender is assigned a number of receivers. The number of receivers can be either: *uniform* from one up to the number of neighbors in the network; *high*, involving at least 75% of the neighbors; or *low*, involving at most 25% of the neighbors.

- *Degree of Payload Sharing.* Each rule-sender is assigned a degree of payload sharing. The payload sharing can be *uniform*, from one to the number of receivers for the rule-sender; *high*, involving at least 75% of the receivers; or *low*, involving at most 25% of the receivers. Note the preceeding parameter, number of receivers, inherently limits the maximum degree of payload sharing.

The following are the optimization combinations that we tested. An abbreviation for the combination is shown in parenthesis.

- No Optimization ($\emptyset$).
- `Redirection-via-Intermediary` Only ($\mathcal{R}$).
- `Broadcast-or-Unicast` Only ($\mathcal{B}$).
- `LPL-Tuner` Only ($\mathcal{L}$).
- `Redirection-via-Intermediary` and `LPL-Tuner` Jointly ($\mathcal{R}$+$\mathcal{L}$).
- `Broadcast-or-Unicast` and `LPL-Tuner` Jointly ($\mathcal{B}$+$\mathcal{L}$).
- All Three Independently ($\mathcal{R}/\mathcal{B}/\mathcal{L}$).
- All Three Jointly ($\mathcal{R}$+$\mathcal{B}$+$\mathcal{L}$).

In the combinations that did not involve `LPL-Tuner`, we assigned a single coarse grained polling interval for all network nodes by running the LPL optimization with the average data rate across all flows as input (99). Note that we are making a rather generous assumption that even in the No Optimization ($\emptyset$) case, a coarse level of optimization is performed.

The results of testing each set of parameters with each optimization combination is shown in Table 4.4. The numerical values are the magnitude gain over no optimization. Hence, $\emptyset$ is always 1×. Each numerical value is the median of one hundred test trials.

The most striking gain is 17.49× for Setting 2 with $\mathcal{R}$+$\mathcal{B}$+$\mathcal{L}$, and several instances offer gains of one order of magnitude or more. Note that $\mathcal{R}$ and $\mathcal{B}$ are often 1×. This indicates that gains from independent optimizations may not be so striking as to warrant consideration on their own. In one case, optimization even decreases performance very slightly (0.99×), suggesting that isolated optimizations may be working with models that are too simple to model actual performance.

Min, max, 25th and 75th percentile trends were similar. Next we take a closer look at the parameter settings and their impact on optimization.

Setting			*Optimization*							
#	*receivers*	*sharing*	$\emptyset$	$\mathcal{R}$	$\mathcal{B}$	$\mathcal{L}$	$\mathcal{R}$+$\mathcal{L}$	$\mathcal{B}$+$\mathcal{L}$	$\mathcal{R}/\mathcal{B}/\mathcal{L}$	$\mathcal{R}$+$\mathcal{B}$+$\mathcal{L}$
heterogeneous										
1	uniform	uniform	1.00	1.33	1.48	3.62	13.36	3.73	3.62	14.98
2	high	high	1.00	1.51	3.79	3.00	16.26	5.28	3.81	17.49
3	high	low	1.00	1.00	0.99	2.74	9.95	2.69	2.24	9.95
4	low	*	1.00	1.00	1.00	3.68	9.00	3.51	4.25	8.21
homogeneous										
5	uniform	uniform	1.00	1.00	1.71	3.07	3.07	3.53	3.51	3.53
6	high	high	1.00	1.00	4.77	3.15	3.17	5.31	4.64	5.31
7	high	low	1.00	1.00	1.00	2.94	2.94	3.36	3.43	3.36
8	low	*	1.00	1.00	1.00	2.80	2.80	2.80	2.69	2.80

Table 4.4: Energy savings relative to no optimization ($\emptyset$).

Homogeneous vs. Heterogeneous Node Energy Levels

As a general rule, heterogeneous environments have a lot more to gain from optimization than homogeneous environments. This is mostly due to the fact that many protocols such as LPL are not designed with node heterogeneity in mind. Presently, we focus on node energy-level heterogeneity. In every comparable parameter configuration (Setting 1 vs. 5, 2 vs. 6, 3 vs. 7, 4 vs. 8), heterogeneous environments have more to gain from optimization than homogeneous environments. Specifically, in heterogeneous environments, energy-poor senders and receivers can pass on a good deal of the communication burden to energy-rich neighbors by combining intermediary selection and LPL tuning. For example, Settings 1, 2, 3 and 4 derive a substantial benefit from $\mathcal{R}$+$\mathcal{L}$, whereas the benefit from either $\mathcal{R}$ or $\mathcal{L}$ alone is much less. $\mathcal{R}$+$\mathcal{L}$ is in fact quite close to $\mathcal{R}$+$\mathcal{B}$+$\mathcal{L}$ in these settings, and even besting $\mathcal{R}$+$\mathcal{B}$+$\mathcal{L}$ in Setting 4. This is due to the greedy nature of Listing 4.2. Moreover, $\mathcal{R}$+$\mathcal{L}$ handidly beats $\mathcal{R}/\mathcal{B}/\mathcal{L}$ in each case. This strongly argues for joint optimization.

On the other hand, energy savings are much less when node energy levels are homogeneous. Here, we cannot take advantage of resource heterogeneity, but we can still expose workload heterogeneity (who is sending to whom and at what data rate) to our benefit; idle nodes can share in the communication burden with frequently communicating nodes.

Number of Receivers and Payload Sharing

A rule-sender's number of receivers and degree of payload sharing significantly impact the optimization gains possible. For instance, compare Settings 2 vs. 3 which differ only in the degree of payload sharing. On a range of optimizations, Setting 2 typically outperformed Setting 3 by 51-76%. This is because `Redirection-via-Intermediary` and `Broadcast-or-Unicast` benefit substantially from high payload sharing, and the ability to perform sender-side payload aggregation. In fact, the overall highest gains are in Setting 2 with $\mathcal{R}$+$\mathcal{B}$+$\mathcal{L}$, precisely when there is a high degree of payload sharing combined with joint optimization.

Payload sharing is important for homogeneous settings as well. Optimizations in Setting 6 outperform those in Settings 5, 7 and 8. One interesting note here is that payload sharing in homogeneous settings benefits `Broadcast-or-Unicast` much more than `Redirection-via-Intermediary`. Setting 6's $\mathcal{B}$ and $\mathcal{B}$+$\mathcal{L}$ perform quite closely to $\mathcal{R}$+$\mathcal{B}$+$\mathcal{L}$, suggesting that the optimal choice is often to broadcast when payloads are heavily shared. This contrasts with the heterogeneous case, where the better choice is to redirect via an intermediary.

A rule-sender's number of receivers is an upper limit on the degree of payload sharing. Therefore, when the number of receivers is low, we see very similar performance to when the number of receivers is high and there is low sharing *i.e.,* in Settings 3 vs. 4, and 7 vs. 8.

Reserve Nodes

In the experiments discussed thus far, the senders and receivers for each rule-sender were selected from among all available nodes. We next experimented with homogeneous settings where one node is kept in reserve *e.g.,* for the sole purpose of serving as an intermediary. Table 4.5 shows the result of joint optimization when the reserve node's energy level is varied. Reserve nodes at 2× and even 10× the energy level of other network nodes do not provide significant benefit. However, at 100×, the benefits of the reserve node are pronounced. Energy levels often vary dramatically in practice (*e.g.,* AA batteries vs. solar-charged car battery).

Reserve Node Energy Relative to Other Nodes	$\mathcal{R}$+$\mathcal{B}$+$\mathcal{L}$ gain over $\emptyset$
1×	3.67
2×	3.93
10×	4.35
100×	13.06

Table 4.5: Energy savings of including one reserve node relative to no optimization.

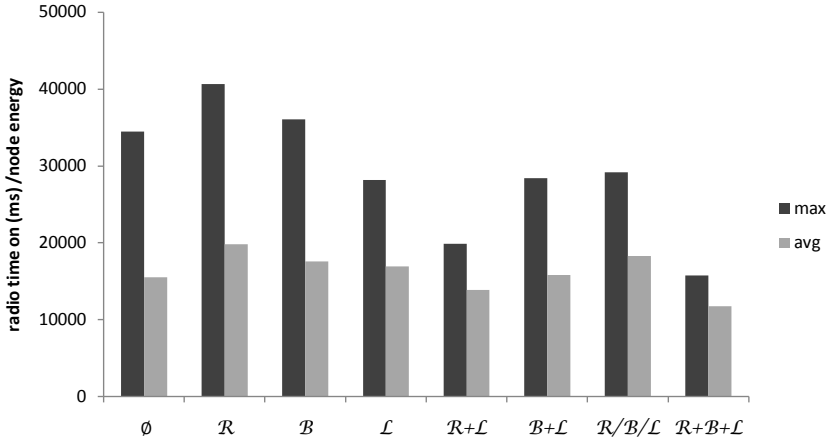


Figure 4.3: `wireless-netopt` performance on six node mesh testbed.

These findings suggest it is better to use `wireless-netopt` at a coarse granularity (*e.g.,* per-deployment) rather than as a fine-tuning measure.

4.3.2 Testbed

Our testbed consisted of six Tmote devices in the same local area. Three nodes took the role of energy-rich nodes, with 1 at 100× base and 2 at 10× base. We ran a simple program with one distributed rule shown in Section 4.2. For the MAC, we used BMAC+, an implementation of LPL for the Tmote platform. It is summarized in Section 4.2.3 and discussed in (103). We generated four rule-senders for testing, each with a different sender. The number of receivers and degree of payload sharing were chosen uniformly, mirroring Setting 1 in Table 4.4. To arrive at energy usage, we measured active radio time. We opted not to construct detailed

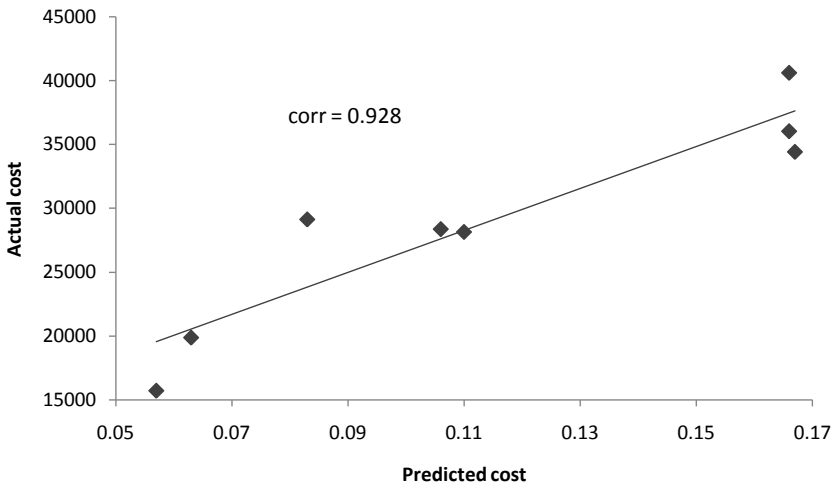


Figure 4.4: Actual testbed performance correlates strongly with predicted performance.

records of times in various radio modes because the power draw for Carrier Sense, Transmit, Receive and Polling modes are all very similar for the Tmote platform. Conversely, Sleep mode is very insignificant for this platform. It was not within our capabilities to accurately measure energy usage directly for six nodes simultaneously. As a sanity check, we also made sure that the message delivery ratio to receivers was the same under all optimizations.

Figure 4.3 shows the effect of applying optimizations to minimize active radio time normalized by node energy. The "max" series indicates the normalized active radio time with respect to our primary objective, minimizing the maximum normalized energy drain. The "avg" series indicates the average normalized energy drain. We discuss the "max" series first.

The radio time trends match the energy savings trends that we saw in simulation. $\mathcal{R}$+$\mathcal{B}$+$\mathcal{L}$ is able to save 2.2× in active radio time compared to $\emptyset$. The magnitude of gains are not as big here because of two factors. First, we used only a single data rate, .1 packets per second, for all flows. This caused the data rate for the default LPL setting to be quite accurate to begin with, since we used the generous setting of the average of all flow data rates. We would expect to see a greater magnitude of gain had we used more varied data rates. Second, a six node network has less chances for payload sharing than a ten node network simply due to its size. This

subsequently cuts down on the magnitude of benefit for both `Redirection-via-Intermediary` and `Broadcast-or-Unicast`.

Interestingly, both $\mathcal{R}$ and $\mathcal{B}$ perform worse than $\emptyset$. This is because $\mathcal{R}$ and $\mathcal{B}$ use a basic model for message transmission and reception costs (discussed in Section 4.2) that does not take into account MAC interactions. This mirrors the findings in Table 4.4 where $\mathcal{R}$ and $\mathcal{B}$ alone rarely provide the best gains. When an accurate MAC model is used (as in $\mathcal{R}$+$\mathcal{L}$ and $\mathcal{B}$+$\mathcal{L}$), the energy savings improve significantly.

The "avg" series followed the same trend as the "max" series. This is convenient outcome because it means that as a side benefit, minimizing the maximum energy drain also yields acceptable average energy drains.

Figure 4.4 shows how well the optimizer did at predicting actual runtime costs. It turns out that the optimizer's predicted costs are highly correlated with the actual costs with a correlation value of 0.928. This means that the optimizer is able to accurately gauge the magnitude of actual performance gain. This result is a pleasant surprise since database optimizers are traditionally not well-known for their accuracy at modeling absolute execution costs.

4.4 Summary

Wireless communication is the most power-intensive operation of modern embedded and networked systems, and can dominate other operations by an order of magnitude or more. Many clever optimization schemes have emerged to minimize communication, and these schemes vary widely in terms of positioning within the networking stack. Unfortunately, many of these schemes are not used by programmers because programmers often need to understand, and then tune the optimizations to best suite their application.

This chapter looked at taking three of these optimizations and applying them automatically to program source. We found that automatic optimization is indeed quite practical within the declarative programming model because programs are very easy to analyze and rewrite. Furthermore, when combinations of optimizations are considered, joint optimization

often outperforms a series of independent optimizations. The energy savings can be over 2× in heterogeneous node settings.

Chapter 5

Related Work

5.1 Declarative Sensor Networks

Numerous deployment experiences have demonstrated that developing low-level software for sensor nodes is very difficult (6; 106). This challenge has led to a large body of work exploring high-level programming models that capture application semantics in a simple fashion. By borrowing languages from other domains, these models have demonstrated that powerful subsets of requisite functionality can be easily expressed. TinyDB showed that the task of requesting data from a network can be written via declarative, SQL-like queries (31) and that a powerful query runtime has significant flexibility and opportunity for automatic optimization. Abstract regions (35) and Kairos (36) showed that data-parallel reductions can capture aggregation over collections of nodes, and that such programs have a natural trade off between energy efficiency and precision. SNACK (38), Tenet (10), Regiment (37) and Flask (39) demonstrated that a dataflow model allows multiple data queries to be optimized into a single efficient program. Following the same multi-tier system architecture of Tenet, semantic streams (107) showed that a coordinating base station can use its knowledge of the network to optimize a declarative request into sensing tasks for individual sensors.

From these efforts it appears that declarative, data-centric languages are a natural fit for

many sensor network applications. But these works typically focus on data gathering and processing, leaving many core networking issues to built-in library functions. Tenet even takes the stance that applications should not be introducing new protocols, delegating complex communication to resource-rich higher-level devices. Our goal in DSN is to more aggressively apply the power of high-level declarative languages in sensornets to all levels of a system's data acquisition, the networking logic involved in communicating that data, and the management of key system resources, while retaining architectural flexibility.

In the Internet domain, the P2 project (29; 30; 73) demonstrated that declarative logic languages can concisely describe many Internet routing protocols and overlay networks. Furthermore, the flexibility the language gives to the runtime for optimization means that these high-level programs can execute efficiently.

DSN takes these efforts and brings them together, defining a declarative, data-centric language for describing data management and communication in a wireless sensor network. From P2, DSN borrows the idea of designing a protocol specification language based on the recursive query language Datalog. Sensornets have very different communication abstractions and requirements than the Internet, however, so from systems such as ASVMs (40), TinyDB (31), and VM* (41), DSN borrows techniques and abstractions for higher-level programming in the sensornet domain. Unlike these prior efforts, DSN pushes declarative specification through an entire system stack, touching applications above and single-hop communication below, and achieves this in the kilobytes of RAM and program memory typical to sensor nodes today.

5.2 Rendezvous and Proxy Selection

Related work stems from both networking and databases.

5.2.1 Network Protocol Optimization

Prior work in network protocol optimization generally focuses on packet processing performance on the single node, usually by adapting techniques from general compiler optimization

to increase single-node packet processing performance: inlining, outlining, code cloning, rearranging branches, and IPC to function call conversion (108; 109; 110; 111; 112). On the other hand, our focus is automated multi-node protocol optimization.

Several efforts have attempted to enable greater network flexibility. Active networks research moved aggressively to introduce greater programmability into networks (113). Our work introduces a limited amount of network reprogramming, driven by optimizer decisions rather than node-level code injections. Like our work, i3 identifies rendezvous and proxy selection as fundamental to network design, and provides great flexibility for their selection (76). Unlike our work, i3 does not aim to optimize rendezvous and proxy selection from program source.

5.2.2 Database Query Optimization

The network optimization mechanisms introduced in this work can be viewed as generalizations of query processing and optimization mechanisms familiar to the database community. Changing rendezvous is conceptually very similar to reordering database join operations. System R popularized the ideas of optimizing join ordering with respect to disk IO, CPU, and table statistics (81). Like (114), the current work fundamentally adopts the same optimization framework, extended to the networked setting. We significantly broaden the scope of what can be reordered, and thus what reordering is capable of by viewing "application data" and "networking data" under the same lens.

In the past, deductive database query optimization focused on combining "push" with "pull" query processing (69). The main result was the Magic Sets algorithm that transforms programs to take advantage of the benefits of pull processing while executing in a push context (71). The work of (29) extended this to the networked setting, specifically applying an entirely pull processing approach to the example of routing as in Section 3.4.2. In contrast, this work suggests that hybrids using a mixture of push and pull offer the best cost for many practical networking scenarios.

We suspect it is possible to generalize MiM Rewrite to all recursion, just as algorithms for LR have been subsumed by the Magic Sets algorithm (69). However, our experience indicates

that LR is the most common recursion, especially in networking. This also echoes the remarks of (69) for traditional Datalog.

5.3 Cross-Layer Optimization

Cross-layer network optimization has received considerable attention in the literature. We discuss specific efforts at cross-layer optimization, general cross-layer optimization frameworks, and programmer support for cross-layer optimization.

There is much work on specific cross-layer optimizations in sensornets. For example, the author of (115) investigates the joint optimization of transmission energy and circuit energy, and the joint optimization of MAC, link layer and routing for sensornets. The authors of (116) also looks at cross-layer energy optimization of MAC and routing for sensornets. The authors of (117) reconfigure routing components and radio components at runtime in a flood sensornet monitoring application to optimize for energy consumption and event notification latency. Specific cross-layer optimizations have also been proposed for other networking contexts, such as Internet and Mobile Ad-hoc Networks (MANETs). Future work is to investigate the challenges in integrating these disparate point solution optimizations into `wireless-netopt`.

A lot of work has also attempted to generalize cross-layer optimization. The authors in (118) survey recent work that looks at network layering as optimization decomposition. The spirit of this field is in fact very similar to that of our network optimizer: the entire network and networking stack forms an optimization problem whose objective is provided by the user application. The main lessons of the field are that if convexity holds, it is possible to compute globally optimal solutions from local subproblems. Also, much like in our work on three specific optimizations, related subproblems are linked by optimization variables. These optimization variables form the interface between layers. The work in (119) provides a tutorial on advances in cross-layer optimization for wireless single-hop networks, and describes several challenges to extending the work to multi-hop networks. The approach uses convex optimization as an important tool. A main goal in that work is to have loosely coupled layers, which translates

into few shared optimization variables between layers. Unlike our work, this field has focused primarily on formalizing the optimization problem, rather than on programmer support for realizing the automatic application of the optimizations.

Several efforts have looked at making cross-layer variables more visible to programmers. A representative work is (120), which lets nesC programmers annotate certain variables as shared across component boundaries. Layers can coordinate usage of the shared data, and save on storage space by only maintaining a single copy of the variables. Conceptually, it would be possible to perform joint optimization in this model. However, unlike `wireless-netopt`, the burden is on the programmer to configure data sharing, as well as optimizations.

Chapter 6

Discussion

Based on our investigation thus far, the benefits of declarative programming appear substantial. However, significant innovation is still needed before automated network optimization becomes commonplace. Technical challenges ahead can be clustered with respect to key participants in the networking ecosystem: declarative programmers; optimization designers; and external developers.

6.1 Declarative Programmers

DSN has served declarative programmers fairly well, from those that want to simply pose SQL-like queries, to those that wish to develop new services at all layers of the system down to the device drivers. However, the initial language learning curve is not trivial. To aid accessibility, the power of declarative programming needs to be reconciled with the familiarity of imperative programming. One approach is to more tightly integrate imperative and declarative programming, as the Microsoft LINQ project has done (121). A much more ambitious alternative approach is to bring the ideas of network optimization to widely used imperative languages such as Java or Python. This avenue is appealing because it would benefit a large

programmer population. At the same time, the challenge is sizable, as tractability concerns for program analysis arise (122; 123).

6.2 Optimization Designers

`netopt` aids optimization designers by providing an architecture for automatic optimization. Currently, the compile-time optimization decreases the per-deployment manual optimization effort. Opportunities may also arise for adaptivity to environmental dynamics during a deployment when environment changes occur frequently; environment state is uncertain or obscured to the optimizer; or visibility of workload and network state changes are delayed to the optimizer.

One way to incorporate adaptivity is to replan execution by iteratively running the one-time centralized optimization process. The main advantage is that the optimization procedures port naturally from the static optimization setting. Additionally, replanning retains its optimality with respect to the environment snapshot. The main disadvantage is that as frequency of execution increases, so does the overhead of snapshot gathering and optimization parameter redistribution. Additionally, partially obscured or time delayed environment state is still unresolved.

An alternative to more frequent centralized planning is optimizing directly for adaptivity (124; 125). For the rendezvous and proxy optimizations as well as some of the cross-layer optimizations, we have sketched initial rewrites that emit programs that embed dynamic optimization logic. By making decisions in a distributed and continuous fashion, it may be possible to react faster to changing conditions, and also react to conditions not visible in a timely manner at the central planner. The possible disadvantage of this approach is that it is not clear that distributed (often greedy) decisions lead to optimal execution, The subtle connections between the fields of databases and networking may again offer insight since similar trade-offs were encountered in earlier work on continuous query processing adaptivity.

Extending `netopt` in another way may also appeal to the optimization designers. In many

scenarios, two program executions may both satisfy the user even though the exact query response may differ. For example, probabilistic approximate responses can be as acceptable as exact responses while offering significant performance gains (126). Lossy data compression may be as good as lossless compression. Best-effort communication may be as good as reliable communication. These disparate scenarios indicate that it will be formidable to construct a general architecture to support this flexibility. To start, query semantics need to encompass additional constraints that bound navigation within the query response space. Then, we need to outline program equivalence classes for programs that provide similar properties in some dimensions but differ in others. The last step is to build out `netopt` architectural interfaces to support the variety of existing optimized point solutions already in the community. The main challenge is scaling the mapping of designers optimizations to equivalence classes. We suspect that this will not be an easy task to perform automatically, and may require additional metadata such as designer annotations. Yet the benefit is that programmers will continue to capture gains from automatic optimization of their programs.

6.3 Developers of External Systems

`netopt` needs to continue to demonstrate interoperability with external systems. Many DSN users apply the declarative language selectively. That is, they enjoy the flexibility of calling out from DSN to native code. Yet interconnected networked systems must consider inter-program interoperability as well. Inter-program interoperability can be thought of in terms of packets, execution and state, and the declarative model does not seem to be a hindrance to interoperability in these areas. In fact, the model may even ease interoperability by natively supporting proxies, a common interop mechanism.

In terms of packet interoperability, database schema definitions can serve the role of packet format definitions. Schemas specify field layouts and data types for relations, and since packets are tuples in our programming model, the application is straightforward. In terms of execution interoperability, it is the case that a DSN program and traditional program that do not understand each other's protocols will not be able to interoperate. However, this is no different

than the current incompatibility between TCP and UDP, or between any other two protocols not designed with the same properties in mind.

A valid concern is state interoperability. That is, the optimization process changes node-to-state mappings. External systems (e.g. under separate administrative control or legacy nodes) wishing to interoperate may receive incomplete or unexpected protocol state. Manually optimized programs also face this issue when desiring to interoperate. The natural solution is to bridge the two incompatible protocols via proxies, for which `netopt` is already well suited. Through additional declarative "hint" statements, `netopt` might let users influence or constrain the placement of proxies where interfacing with external systems occurs. This offers a path for incremental deployment on networks with external systems, while preserving the benefit of automatic optimization. It very much follows the spirit of manually engineered solutions to interoperability: use of proxies (127).

Chapter 7

Conclusion

Applications and their workloads, as well as networks and their client devices, are continuing to evolve at healthy rates. As they do, we would like to see new combinations of applications and networks work together efficiently. The driving theme of this dissertation has been to ask whether it is possible to achieve respectable application efficiency, while foregoing much of the manual engineering that is the state of the art in building networked systems today.

The field of data management greatly inspired our work. From it, we saw the appeal of taking a declarative approach to programming networked systems. Namely, it might be possible for programmers to succinctly express what they want, rather than how they want it, and; efficient execution would be the responsibility of the programming system, and not the programmer.

To explore the extent of these possibilities, we developed DSN, `netopt` and `wireless-netopt`: a declarative programming environment targeted at networked and embedded systems. This set of compiler, optimizer and runtime simplifies the task of the sensornet programmer in two primary ways. First, programmers write very concise declarative programs that perform quite comparably to hand-crafted imperative implementations, and in one case even matched original pseudocode nearly line-for-line. We demonstrated that a wide spectrum of interesting sensornet problems, as well as a full system stack implementation can be implemented in our

declarative programming language. Second, program source is automatically analyzed and optimized on the programmer's behalf. The optimizations target design decisions that occur frequently in general networking and wireless networking. Programs often receive one order of magnitude, and up to two orders of magnitude benefit versus unoptimized variants. Most importantly, the programmer need not invest added effort to realize these gains.

Ultimately, we found strong opportunities to mold ideas from the field of data management into new applications in the field of networked systems. This even led us to shape network engineering as a form of query optimization. These rich relationships are our main thematic contribution.

Bibliography

[1] Terzis A, Anandarajah A, Moore K, Wang IJ (2006) Slip surface localization in wireless sensor networks for landslide prediction. In: IPSN '06: Proceedings of the 5th international conference on information processing in sensor networks. New York, NY, USA: ACM, pp. 109–116.

[2] Bonnet P, Leopold M, Madsen K (2006) Hogthrob: towards a sensor network infrastructure for sow monitoring (wireless sensor network special day). In: DATE '06: Proceedings of the conference on Design, automation and test in Europe. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, pp. 1109–1109.

[3] Kim S, Pakzad S, Culler D, Demmel J, Fenves G, et al. (2007) Health monitoring of civil infrastructures using wireless sensor networks. In: IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks. New York, NY, USA: ACM, pp. 254–263.

[4] Chu D, Zhao F, Liu J, Goraczko M (2008) Que: A sensor network rapid prototyping tool with application experiences from a data center deployment. In: EWSN '08: Proceeedings of the Fifth European Workshop on Wireless Sensor Networks. pp. 337–353.

[5] Hartung C, Han R, Seielstad C, Holbrook S (2006) Firewxnet: a multi-tiered portable wireless system for monitoring weather conditions in wildland fire environments. In: MobiSys '06: Proceedings of the 4th international conference on Mobile systems, applications and services. New York, NY, USA: ACM, pp. 28–41.

[6] Werner-Allen G, Lorincz K, Johnson J, Lees J, Welsh M (2006) Fidelity and yield in a volcano monitoring sensor network. In: OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation. Berkeley, CA, USA: USENIX Association, pp. 27–27.

[7] Vasilescu I, Kotay K, Rus D, Dunbabin M, Corke P (2005) Data collection, storage, and retrieval with an underwater sensor network. In: SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems. New York, NY, USA: ACM, pp. 154–165.

[8] Barrenetxea G, Ingelrest F, Schaefer G, Vetterli M (2008) The hitchhiker's guide to successful wireless sensor network deployments. In: SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems. New York, NY, USA: ACM, pp. 43–56.

[9] Sugihara R, Gupta RK (2008) Programming models for sensor networks: A survey. ACM Transactions on Sensor Networks (TOSN) 4:1–29.

[10] Gnawali O, Jang KY, Paek J, Vieira M, Govindan R, et al. (2006) The tenet architecture for tiered sensor networks. In: SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems. New York, NY, USA: ACM, pp. 153–166.

[11] Buonadonna P, Gay D, Hellerstein J, Hong W, Madden S (2005) Task: sensor network in a box. In: EWSN '05: Proceeedings of the Second European Workshop on Wireless Sensor Networks. pp. 133–144.

[12] Levis P, Culler D (2002) Maté: a tiny virtual machine for sensor networks. In: ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems. New York, NY, USA: ACM, pp. 85–95.

[13] Ramakrishnan R, Gehrke J (2002) Database Management Systems. McGraw-Hill Science/Engineering/Math.

[14] Condie T, Chu D, Hellerstein JM, Maniatis P (2008) Evita raced: metacompilation for

declarative networks. In: VLDB '08: Proceedings of the 34th international conference on Very large data bases. VLDB Endowment, pp. 1153–1165.

[15] Chudak FA, Shmoys DB (1999) Improved approximation algorithms for a capacitated facility location problem. In: SODA '99: Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, pp. 875–876.

[16] Abadi DJ, Madden S, Lindner W (2005) Reed: robust, efficient filtering and event detection in sensor networks. In: VLDB '05: Proceedings of the 31st international conference on Very large data bases. VLDB Endowment, pp. 769–780.

[17] Freedman MJ, Freudenthal E, Mazières D (2004) Democratizing content publication with coral. In: NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation. Berkeley, CA, USA: USENIX Association, pp. 18–18.

[18] Chandramouli B, Xie J, Yang J (2006) On the database/network interface in large-scale publish/subscribe systems. In: SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data. New York, NY, USA: ACM, pp. 587–598.

[19] Shieh A, Myers AC, Sirer EG (2005) Trickles: a stateless network stack for improved scalability, resilience, and flexibility. In: NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation. Berkeley, CA, USA: USENIX Association, pp. 175–188.

[20] Stoica I (2000) Stateless Core: A Scalable Approach for Quality of Service in the Internet. Ph.D. thesis, Carnegie Mellon University.

[21] Sherman A, Lisiecki PA, Berkheimer A, Wein J (2005) Acms: the akamai configuration management system. In: NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation. Berkeley, CA, USA: USENIX Association, pp. 245–258.

[22] Ramasubramanian V, Haas ZJ, Sirer EG (2003) Sharp: a hybrid adaptive routing protocol for mobile ad hoc networks. In: MobiHoc '03: Proceedings of the 4th ACM international symposium on Mobile ad hoc networking & computing. New York, NY, USA: ACM, pp. 303–314.

[23] Mohan P, Padmanabhan VN, Ramjee R (2008) Nericell: rich monitoring of road and traffic conditions using mobile smartphones. In: SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems. New York, NY, USA: ACM, pp. 323–336.

[24] Lampson B (2003) Getting computers to understand. Journal of the ACM (JACM) 50:70–72.

[25] Whaley J, Lam MS (2004) Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation. New York, NY, USA: ACM, pp. 131–144.

[26] Becker MY, Sewell P (2004) Cassandra: Distributed Access Control Policies with Tunable Expressiveness. In: 5th IEEE International Workshop on Policies for Distributed Systems and Networks.

[27] Abiteboul S, Abrams Z, Haar S, Milo T (2005) Diagnosis of asynchronous discrete event systems: datalog to the rescue! In: PODS '05: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. New York, NY, USA: ACM, pp. 358–367.

[28] Singh A, Maniatis P, Roscoe T, Druschel P (2006) Using queries for distributed monitoring and forensics. In: EuroSys '06: Proceedings of the 2006 EuroSys conference. New York, NY, USA: ACM, pp. 389–402.

[29] Loo BT, Hellerstein JM, Stoica I, Ramakrishnan R (2005) Declarative routing: extensible routing with declarative queries. In: SIGCOMM '05: Proceedings of the 2005 conference

on Applications, technologies, architectures, and protocols for computer communications. New York, NY, USA: ACM, pp. 289–300.

[30] Loo BT, Condie T, Hellerstein JM, Maniatis P, Roscoe T, et al. (2005) Implementing declarative overlays. In: SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles. New York, NY, USA: ACM, pp. 75–90.

[31] Madden SR, Franklin MJ, Hellerstein JM, Hong W (2005) Tinydb: an acquisitional query processing system for sensor networks. ACM Trans Database Syst 30:122–173.

[32] Levis P, Patel N, Culler D, Shenker S (2004) Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In: NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation. Berkeley, CA, USA: USENIX Association, pp. 15–28.

[33] Rao A, Papadimitriou C, Shenker S, Stoica I (2003) Geographic routing without location information. In: MobiCom '03: Proceedings of the 9th annual international conference on Mobile computing and networking. New York, NY, USA: ACM Press, pp. 96–108.

[34] Oh S, Chen P, Manzo M, Sastry S (2006) Instrumenting wireless sensor networks for real-time surveillance. In: Proc. of the International Conference on Robotics and Automation.

[35] Welsh M, Mainland G (2004) Programming sensor networks using abstract regions. In: NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation. Berkeley, CA, USA: USENIX Association, pp. 29–42.

[36] Gummadi R, Kothari N, Govindan R, Millstein T (2005) Kairos: a macro-programming system for wireless sensor networks. In: SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles. New York, NY, USA: ACM Press, pp. 1–2.

[37] Newton R, Welsh M (2004) Region streams: functional macroprogramming for sensor networks. In: DMSN '04: Proceeedings of the 1st international workshop on Data management for sensor networks. New York, NY, USA: ACM Press, pp. 78–87.

[38] Greenstein B, Kohler E, Estrin D (2004) A sensor network application construction kit (snack). In: SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems. New York, NY, USA: ACM Press, pp. 69–80.

[39] Mainland G, Morrisett G, Welsh M (2008) Flask: staged functional programming for sensor networks. In: ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming. New York, NY, USA: ACM, pp. 335–346.

[40] Levis P, Gay D, Culler D (2005) Active sensor networks. In: NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation. Berkeley, CA, USA: USENIX Association, pp. 343–356.

[41] Koshy J, Pandey R (2005) Vmstar: synthesizing scalable runtime environments for sensor networks. In: SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems. New York, NY, USA: ACM, pp. 243–254.

[42] Ramakrishnan R, Ullman JD (1993) A survey of research on deductive database systems. Journal of Logic Programming 23:125–149.

[43] Gay D, Levis P, von Behren R, Welsh M, Brewer E, et al. (2003) The nesc language: A holistic approach to networked embedded systems. In: PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation. New York, NY, USA: ACM, pp. 1–11.

[44] Dsn programming tutorial. Http://db.cs.berkeley.edu/dsn.

[45] Fonseca R, Ratnasamy S, Zhao J, Ee CT, Culler D, et al. (2005) Beacon vector routing: scalable point-to-point routing in wireless sensornets. In: NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation. Berkeley, CA, USA: USENIX Association, pp. 329–342.

[46] Ee CT, Ratnasamy S, Shenker S (2006) Practical data-centric storage. In: NSDI'06: Proceedings of the 3rd conference on Networked Systems Design & Implementation. Berkeley, CA, USA: USENIX Association, pp. 24–24.

[47] Karp B, Kung HT (2000) GPSR: greedy perimeter stateless routing for wireless networks. In: Mobile Computing and Networking. pp. 243–254.

[48] Woo A, Culler D (2003) Evaluation of efficient link reliability estimators for low-power wireless networks. Technical Report UCB/CSD-03-1270, EECS Department, University of California, Berkeley.

[49] Madden S, Franklin MJ, Hellerstein JM, Hong W (2002) Tag: a tiny aggregation service for ad-hoc sensor networks. In: OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation. New York, NY, USA: ACM, pp. 131–146.

[50] Leong B, Liskov B, Morris R (2006) Geographic routing without planarization. In: NSDI'06: Proceedings of the 3rd conference on Networked Systems Design & Implementation. Berkeley, CA, USA: USENIX Association, pp. 25–25.

[51] Polastre J, Hui J, Levis P, Zhao J, Culler D, et al. (2005) A unifying link abstraction for wireless sensor networks. In: SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems. New York, NY, USA: ACM Press, pp. 76–89.

[52] Zeng H, Ellis CS, Lebeck AR, Vahdat A (2003) Currentcy: A unifying abstraction for expressing energy management policies. In: USENIX Annual Technical Conference, General Track. pp. 43–56.

[53] Jiang X, Taneja J, Ortiz J, Tavakoli A, Dutta P, et al. (2007) An architecture for energy management in wireless sensor networks. In: WSNA '07: International Workshop on Wireless Sensor Network Architecture.

[54] Tinyos. Http://www.tinyos.net.

[55] Polastre J, Szewczyk R, Culler D (2005) Telos: enabling ultra-low power wireless research. In: IPSN.

[56] Hellerstein JM, Stonebraker M (2005) Anatomy of a database system. Readings in Database Systems, 4th Edition .

[57] Kohler E, Morris R, Chen B, Jannotti J, Kaashoek MF (2000) The click modular router. ACM Transactions on Computer Systems 18:263–297.

[58] Omega testbed. Http://omega.cs.berkeley.edu.

[59] Levis P, Lee N, Welsh M, , Culler D (2003) Tossim: Accurate and scalable simulation of entire tinyos applications. In: SenSys '03: In Proceedings of the First ACM Conference on Embedded Networked Sensor Systems.

[60] Tolle G, Culler D (2005) Design of an application-cooperative management system for wireless sensor networks. In: EWSN '05: Proceeedings of the Second European Workshop on Wireless Sensor Networks.

[61] Chu D, Tavakoli A, Popa L, Hellerstein JM (2006) Entirely declarative sensor network systems. In: VLDB '06: Proceedings of the Thirty Second International Conference on Very Large Data Bases.

[62] Ee CT, Fonseca R, Kim S, Moon D, Tavakoli A, et al. (2006) A modular network layer for sensornets. In: OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation. Berkeley, CA, USA: USENIX Association, pp. 18–18.

[63] Krishnamurthy R, Ramakrishnan R, Shmueli O (1988) A framework for testing safety and effective computability of extended datalog. In: SIGMOD '88: Proceedings of the 1988 ACM SIGMOD international conference on Management of data. pp. 154–163.

[64] Ong J, Fogg D, Stonebraker M (1983) Implementation of data abstraction in the relational database system ingres. SIGMOD Rec 14:1–14.

[65] Franklin M, Zdonik S (1998) Data in your face: push technology in perspective. In: SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data. New York, NY, USA: ACM, pp. 516–519.

[66] Akamai. Http://www.akamai.com.

[67] Limelight. Http://www.limelightnetworks.com.

[68] Bernstein D. Syn cookies. Http://cr.yp.to/syncookies.html.

[69] Ullman JD (1989) Principles of Database and Knowledge-Base Systems. Volume 2, The New Technologies. Computer Science Press.

[70] Abiteboul S, Hull R, Vianu V (1995) Foundations of Databases. Computer Science Press.

[71] Beeri C, Ramakrishnan R (1987) On the power of magic. In: PODS '87: Proceedings of the sixth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems. New York, NY, USA: ACM, pp. 269–284.

[72] Bernstein PA, Goodman N, Wong E, Reeve CL, Jr JBR (1981) Query processing in a system for distributed databases (sdd-1). ACM Trans Database Syst 6:602–625.

[73] Loo BT, Condie T, Garofalakis M, Gay DE, Hellerstein JM, et al. (2006) Declarative networking: language, execution and optimization. In: SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data. New York, NY, USA: ACM, pp. 97–108.

[74] Seshadri P, Hellerstein JM, Pirahesh H, Leung TYC, Ramakrishnan R, et al. (1996) Cost-based optimization for magic: algebra and implementation. SIGMOD Rec 25:435–446.

[75] Johnson DB, Maltz DA (1996) Dynamic source routing in ad hoc wireless networks. In: Mobile Computing, Kluwer Academic Publishers, volume 353.

[76] Stoica I, Adkins D, Zhuang S, Shenker S, Surana S (2004) Internet indirection infrastructure. IEEE/ACM Trans Netw 12:205–218.

[77] Aggarwal C, Yu P (2006) A survey of synopsis construction in data streams. Data Streams: Models and Algorithms :169–208.

[78] Estan C, Savage S, Varghese G (2003) Automatically inferring patterns of resource consumption in network traffic. In: SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications. New York, NY, USA: ACM, pp. 137–148.

[79] Xu K, Zhang ZL, Bhattacharyya S (2005) Profiling internet backbone traffic. SIGCOMM CCR 35:169–180.

[80] Qiu L, Padmanabhan VN, Voelker GM (2001) On the placement of web server replicas. In: INFOCOM '01. Proceedings of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. volume 3, pp. 1587–1596.

[81] Selinger PG, Astrahan MM, Chamberlin DD, Lorie RA, Price TG (1979) Access path selection in a relational database management system. In: SIGMOD '79: Proceedings of the 1979 ACM SIGMOD international conference on Management of data. New York, NY, USA: ACM, pp. 23–34.

[82] Mitchell G, Dayal U, Zdonik SB (1993) Control of an extensible query optimizer: A planning-based approach. In: VLDB '93: Proceedings of the 19th International Conference on Very Large Data Bases. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., pp. 517–528.

[83] Graefe G, McKenna WJ (1993) The volcano optimizer generator: Extensibility and efficient search. In: ICDE '93: Proceedings of the Ninth International Conference on Data Engineering.

[84] White B, Lepreau J, Stoller L, Ricci R, Guruprasad S, et al. (2002) An integrated experimental environment for distributed systems and networks. In: SIGOPS Oper. Syst. Rev. New York, NY, USA: ACM, volume 36, pp. 255–270.

[85] Motelab. Http://motelab.eecs.harvard.edu.

[86] Medina A, Lakhina A, Matta I, Byers J (2001) Brite: An approach to universal topology generation. In: MASCOTS '01: Proceedings of the Ninth International Symposium in Modeling, Analysis and Simulation of Computer and Telecommunication Systems. Washington, DC, USA: IEEE Computer Society, p. 346.

[87] Hui JW, Culler D (2004) The dynamic behavior of a data dissemination protocol for network programming at scale. In: SenSys '04: Proceedings of the 2nd international

conference on Embedded networked sensor systems. New York, NY, USA: ACM, pp. 81–94.

[88] Yang J, Soffa ML, Selavo L, Whitehouse K (2007) Clairvoyant: a comprehensive source-level debugger for wireless sensor networks. In: SenSys '07: Proceedings of the 5th international conference on Embedded networked sensor systems. New York, NY, USA: ACM, pp. 189–203.

[89] Cao Q, Abdelzaher T, Stankovic J, He T (2007) An interactive unix shell for low-end sensor nodes with liteos. In: SenSys '07: Proceedings of the 5th international conference on Embedded networked sensor systems. New York, NY, USA: ACM, pp. 381–382.

[90] Chandramouli B, Xie J, Yang J (2006) On the database/network interface in large-scale publish/subscribe systems. In: SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data. New York, NY, USA: ACM, pp. 587–598.

[91] Chandramouli B, Yang J (2008) End-to-end support for joins in large-scale publish/subscribe systems. Proc VLDB Endow 1:434–450.

[92] Sadler CM, Martonosi M (2006) Data compression algorithms for energy-constrained devices in delay tolerant networks. In: SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems. New York, NY, USA: ACM, pp. 265–278.

[93] Klues K, Hackmann G, Chipara O, Lu C (2007) A component-based architecture for power-efficient media access control in wireless sensor networks. In: SenSys '07: Proceedings of the 5th international conference on Embedded networked sensor systems. New York, NY, USA: ACM, pp. 59–72.

[94] Yang GZ (2006) Body Sensor Networks. Secaucus, NJ, USA: Springer-Verlag New York, Inc.

[95] Mainwaring A, Culler D, Polastre J, Szewczyk R, Anderson J (2002) Wireless sensor networks for habitat monitoring. In: WSNA '02: Proceedings of the 1st ACM international

workshop on Wireless sensor networks and applications. New York, NY, USA: ACM, pp. 88–97.

[96] Xu N, Rangwala S, Chintalapudi KK, Ganesan D, Broad A, et al. (2004) A wireless sensor network for structural monitoring. In: SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems. New York, NY, USA: ACM, pp. 13–24.

[97] Polastre J, Hill J, Culler D (2004) Versatile low power media access for wireless sensor networks. In: SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems. New York, NY, USA: ACM, pp. 95–107.

[98] Hui JW (2008) An Extended Internet Architecture for Low-Power Wireless Networks - Design and Implementation. Ph.D. thesis, EECS Department, University of California, Berkeley.

[99] Ye W, Silva F, Heidemann J (2006) Ultra-low duty cycle mac with scheduled channel polling. In: SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems. New York, NY, USA: ACM, pp. 321–334.

[100] Kotla R, Alvisi L, Dahlin M, Clement A, Wong E (2007) Zyzzyva: speculative byzantine fault tolerance. In: SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles. ACM, pp. 45–58.

[101] Cowling J, Myers D, Liskov B, Rodrigues R, Shrira L (2006) Hq replication: a hybrid quorum protocol for byzantine fault tolerance. In: OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation. Berkeley, CA, USA: USENIX Association, pp. 177–190.

[102] Arora S, Hazan E, Kale S. Multiplicative weights method: a meta-algorithm and its applications. Http://www.cs.princeton.edu/ arora/pubs/MWsurvey.pdf.

[103] Avvenuti M, Corsini P, Masci P, Vecchio A (2006) Increasing the efficiency of preamble sampling protocols for wireless sensor networks. In: MCWC '06: Mobile Computing and Wireless Communications International Conference.

[104] Moteiv. Tmote sky datashett. Http://www.sentilla.com/pdf/eol/tmote-sky-datasheet.pdf.

[105] Whitehouse K, Sharp C, Brewer E, Culler D (2004) Hood: a neighborhood abstraction for sensor networks. In: MobiSys '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services. New York, NY, USA: ACM, pp. 99–110.

[106] Szewczyk R, Polastre J, Mainwaring A, Culler D (2004) Lessons from a sensor network expedition. In: EWSN '04: Proceeedings of the First European Workshop on Wireless Sensor Networks.

[107] Whitehouse K, Liu J, Zhao F (2006) Semantic streams: a framework for composable inference over sensor data. In: EWSN '06: Proceedings of The Third European Workshop on Wireless Sensor Networks.

[108] Castelluccia C, Dabbous W, O'Malley S (1997) Generating efficient protocol code from an abstract specification. IEEE/ACM Trans Netw 5:514–524.

[109] Basu A, Morrisett JG, von Eicken T (1998) Promela++: A language for constructing correct and efficient protocols. In: INFOCOM '01. Proceedings of the 17th Annual Joint Conference of the IEEE Computer and Communications Societies.

[110] Hernek D, Anderson DP (1989) Efficient automated protocol implementation using rtag. Technical Report UCB/CSD-89-526, EECS Department, University of California, Berkeley.

[111] Mosberger D, Peterson LL, Bridges PG, O'Malley S (1996) Analysis of techniques to improve protocol processing latency. In: SIGCOMM Comput. Commun. Rev. New York, NY, USA: ACM, volume 26, pp. 73–84.

[112] Kohler E, Morris R, Chen B (2002) Programming language optimizations for modular router configurations. In: ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems. New York, NY, USA: ACM, pp. 251–263.

[113] Wetherall D (2002) Active network vision and reality: Lessons from a capsule-based system. In: DANCE '02: Proceedings of the 2002 DARPA Active Networks Conference and Exposition. Washington, DC, USA: IEEE Computer Society, p. 25.

[114] Stonebraker M, Aoki PM, Litwin W, Pfeffer A, Sah A, et al. (1996) Mariposa: a wide-area distributed database system. The VLDB Journal 5:048–063.

[115] Cui S (2005) Cross-Layer Optimization in Energy Constrained Networks. Ph.D. thesis, Electrical Engineering, Stanford University.

[116] van LH, Nieberg T, Wu J, Havinga PJ (2004) Prolonging the lifetime of wireless sensor networks by cross-layer interaction. IEEE Wireless Communications 11:78–86.

[117] Grace P, Hughes D, Porter B, Blair GS, Coulson G, et al. (2008) Experiences with open overlays: a middleware approach to network heterogeneity. SIGOPS Oper Syst Rev 42:123–136.

[118] Chiang M, Low S, Calderbank A, Doyle J (2007) Layering as optimization decomposition: A mathematical theory of network architectures. Proceedings of the IEEE 95:255–312.

[119] Lin X, Shroff N, Srikant R (2006) A tutorial on cross-layer optimization in wireless networks. Selected Areas in Communications, IEEE Journal on 24:1452–1463.

[120] Lachenmann A, Marrón PJ, Minder D, Gauger M, Saukh O, et al. (2006) TinyXXL: Language and runtime support for cross-layer interactions. In: Proceedings of the Third Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks. pp. 178–187.

[121] Pialorsi P, Russo M (2007) Introducing microsoft linq. Redmond, WA, USA: Microsoft Press.

[122] Nielson F, Nielson HR, Hankin C (1999) Principles of Program Analysis. Secaucus, NJ, USA: Springer-Verlag New York, Inc.

[123] Tip F (1995) A survey of program slicing techniques. Journal of Programming Languages 3:121–189.

[124] Avnur R, Hellerstein JM (2000) Eddies: continuously adaptive query processing. In: SIGMOD Rec. New York, NY, USA: ACM, volume 29, pp. 261–272.

[125] Deshpande A, Ives Z, Raman V (2007) Adaptive query processing. Found Trends databases 1:1–140.

[126] Garofalakis MN, Gibbon PB (2001) Approximate query processing: Taming the terabytes. In: VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., p. 725.

[127] Walfish M, Stribling J, Krohn M, Balakrishnan H, Morris R, et al. (2004) Middleboxes No Longer Considered Harmful. In: OSDI.

Appendix A

Additional Program Examples

A.1 Link Estimation

The literature indicates that good link estimators are important in wireless environments (48). Several of the examples in Section 2.3 relied upon a built-in link table for managing local neighbors and their link costs. We initially implemented link this way, since it allows us to expose radio hardware-assisted link-estimations *e.g.,* Link Quality Indicator (LQI).

Hardware-independent link estimators provide an alternative to radio-hardware assisted link estimators. These typically use periodic beaconing to calculate packet reception rate (PRR) as an indicator of a neighbor's link cost. Listing A.1 shows the specification of the commonly-used beaconing exponentially weighted moving average (EWMA) link estimator to calculate PRR. The link relation of this program can be used by other programs such as tree routing in place of the built-in link relation.

In this specification, each node periodically sends beacons with sequence numbers (lines 2-4). Upon receiving a beacon, a node updates the PRR by applying EWMA: the previous PRR is weighted against the difference between the last two sequence numbers received (line 7).

The initial link table entries need bootstrapping. The specification accomplishes this by

```
1  % Periodically broadcast beacons
2  timer(@Node, beaconTimer, Period) :- timer(@Node, beaconTimer, Period).
3  seq(@Node, Seq++) :- timer(@Node, beaconTimer, _), seq(@Node, Seq).
4  beacon(@*, Node, Seq) :- seq(@Node, Seq).
5
6  % Estimate link PRR by weighted combination of latest beacon and previous
     PRR
7  link(@Node, Neighbor, NewPrr) :- beacon(@Node, Neighbor, NewSeq),
     lastNeighborSeq(@Node, Neighbor, OldSeq)~, link(@Node, Neighbor, PrrOld)~,
     NewPrr=weight*(NewSeq-OldSeq)+(1-weight)*OldPrr.
8
9  % Update the last neighbor sequence number heard
10 lastNeighborSeq(@Node, Neighbor, Seq) :- link(@Node, Neighbor, PrrNew),
     beacon(@Node, Neighbor, Seq)~.
11
12 % Initialize links that have no previous PRR record
13 link(@Node, Neighbor, initialLinkCost) :- beacon(@Node, Neighbor, Seq),
     -link(@Node, Neighbor, _).
```

Listing A.1: Exponentially Weighted Moving Average Link Estimation

checking new beacons against the *lack* of a corresponding link entry (line 13). DSN employs negation using the dash symbol "-" for checking nonexistence. With minor modifications, the specification can be adapted to variants of time-series link estimation other than EWMA, such as sliding windows.

A.2 Geographic Routing

Geographic routing is a well-studied routing mechanism for sensor networks. Greedy geographic routing sends packets toward the neighbor with the minimum distance to the destination (47). Listing A.2 presents this protocol.

In line 2 neighboring nodes exchange location information. When a message tuple is received, we compute the distances from all neighbors to the destination location and then we

```
1  % Announce own location to neighbors
2  location(@*,Src,SrcX,SrcY) :- location(@Src,Src,SrcX,SrcY).
3
4  % Compute neighbors' distances to message's destination
5  % Function f_distance is the Euclidean distance between the two points
6  computedDistances(@Crt,Nei,DstX,DstY,Dist) :-
     message(@Crt,Src,Dst,DstX,DstY,Data), link(@Crt,Nei,Cost),
     location(@Crt,Nei,NeiX,NeiY), Dist = f_distance(DstX,DstY,NeiX,NeiY), Dst
     != Src.
7
8  % Select the closest neighbor as the next hop
9  shortestCost(@Src,DstX,DstY,<MIN,Dist>) :-
     computedDistances(@Src,Nei,DstX,DstY,Dist).
10 nextHop(@Crt,Next,DstX,DstY) :- shortestCost(@Crt,DstX,DstY,Dist),
     computedDistances(@Crt,Next,DstX,DstY,Dist),
     location(@Crt,Crt,CrtX,CrtY), Dist < f_distance(DstX,DstY,CrtX,CrtY).
11
12 % Invoke fallback routing e.g. right-hand rule or convex hull
13 fallback(@Crt,Xd,Yd,Dist) :- shortestCost(@Crt,DstX,DstY,Dist),
     computedDistances(@Crt,Next,DstX,DstY,Dist),
     location(@Crt,Crt,CrtX,CrtY), Dist >= f_distance(DstX,DstY,CrtX,CrtY).
```

Listing A.2: Geographic Routing

chose as the next hop the node with the smallest distance (line 6-10). The euclidean distance between two coordinates is computed by f_distance. In line 2, the forwarding is straightforward and is similar to Listing 2.2.

Departing from tree routing, geographic routing determines the next hop dynamically on a per-message basis. However, a user is still free to interchange these two routing protocols with only a minimal amount of work since they both ultimately export similar nextHop relations.

Fallback routing, which occurs when the current node responsible for a message is the local minimum but not the end destination, is not shown in the example above but is invoked from line 13. Basic fallback schemes such as planarization (47) are also easy to implement

declaratively. However, more advanced schemes such as (50) require several dozens of rules, due to the algorithm's inherent complexities.

A.3 Localization

The previous example left unanswered how location information is initially established. One option is to provide location as a built-in *e.g.,* as an interface to GPS. Due to cost, most individual nodes are typically not equipped with direct location sensors. A second reasonable option is to manually specify locations with location facts; this is the common case in some deployments. The third option, *localization*, computes node coordinates. Among the many algorithms in this space, NoGeo is is noteworthy for its ability to do without bootstrap location information (33). This service is shown in Listing A.3.

The NoGeo algorithm has three levels of complexity. For ease of exposition, we only present the first which assumes a region's perimeter nodes know their locations. The algorithm proceeds as follows. At short beaconing intervals, neighbors periodically exchange their current locations (line 2). At long estimation intervals, interior nodes compute their own new location estimates by averaging together locations heard from all neighbors (line 5). Perimeter nodes always send the same fixed location estimation and never update their own estimate (line 8). We omit the simple one-line facts that specify the fixed locations of perimeter nodes and initial randomly-estimated locations of interior nodes. After several rounds, nodes' locations converge to points in network connectivity-based coordinate space. The result is a location relation which can be exported for use by other services such as geographic routing in listing A.2.

```
1  % Send current round location to neighbors
2  neighbor(@*,Src,SrcX,SrcY,Round) :-
     timer(@Src,localizationBeaconTimer,Period),
     estimatedLoc(@Src,SrcX,SrcY,Round), round(@Src,Round).
3
4  % Interior nodes average neighbors' locations for updating their own
     location (relaxation step)
5  estimatedLoc(@Src,<AVG,SrcX>,<AVG,SrcY>,Round) :-
     neighbor(@Src,Src,SrcX,SrcY,Round), round(@Src,Round),
     timer(@Src,localizationWindowTimer,_), -perimeter(@Src).
6
7  % Perimeter nodes just refresh their fixed location
8  estimatedLoc(@Src,SrcX,SrcY,Round) :- fixedLoc(@Src,SrcX,SrcY),
     round(@Src,Round), timer(@Src,localizationWindowTimer,_), perimeter(@S).
9
10 % Increase the round after we have computed the new estimate
11 round(@Src,Round++) :- estimatedLoc(@Src,SrcX,SrcY,Round).

13 % Provide the estimated location as the node's location
14 location(@Src,SrcX,SrcY) :- estimatedLoc(@Src,SrcX,SrcY,_).

16 % Set periodic estimation windows and beacon timers
17 timer(@Src,localizationWindowTimer,1000) :-
     timer(@Src,localizationWindowTimer,1000), round(@Src,Round), Round < 1000.
18 timer(@Src,localizationBeaconTimer,200) :-
     timer(@Src,localizationBeaconTimer,200), round(@Src,Round), Round < 1000.
```

Listing A.3: Virtual Coordinates Localization

Appendix B

Additional Rewrite Output

B.1 MiM Rewrite

Listing B.1 shows the result of applying unique variable names to Listing 3.1. This is one of the preliminary steps of MiM Rewrite, as described in Chapter 3.3. Listing B.2 shows the application of MiM Rewrite using the unique variable names of Listing B.1. Rule numbers according to Listing 3.2 are also listed in comments.

Note that there is some redundancy in the attributes of the tuples of $message*$ and $message**$. For example, this redundancy arises in line 14 of Listing B.2 where *interest* attributes $SinkJ$ and $DataK$ are each projecting multiple times to $message*$ attributes. This can lead to bigger than necessary tuples (and hence larger network messages).

It is possible to *prune redundant attributes* to avoid multiple projections of the same attribute with the following observations. Recall that the bound variables for *message* are in the first, third and fourth attributes (m_1, m_3 and m_4) because these attributes join with *interest* in the answer rule. However, according to the recursive rule, only m_1 changes between the head and the body ($CurrentI$ to $NextE$), whereas m_3 and m_4 are directly copied from body to head (m_3 stays as $SinkG$, and m_4 stays as $DataH$). We call m_3 and m_4 *pseudo-free variables* since

```
1  % Prepare for transmission
2  message(@SourceA, SourceA, SinkB, DataC) :-
3    produce(@SourceA, DataC),
4    nexthop(@SourceA, SinkB, NextD).
5
6  % Route message to next hop parent (R1)
7  message(@NextE, SourceF, SinkG, DataH) :-
8   message(@CurrentI, SourceF, SinkG, DataH),
9   nexthop(@CurrentI, SinkG, NextE).
10
11 % Receive if message is of interest (Answer Rule)
12 consume(@SinkJ, DataK) :-
13   message(@SinkJ, SourceM, SinkJ, DataK),
14   interest(@SinkJ, DataK).
15
16 % What is consumed?
17 consume(@Sink, Data)?
```

Listing B.1: Original `BasicProg` with unique variable names applied.

they are bound variables but, like free variables, happen to obey Constraint 1 of Chapter 3.3. It is redundant for $message*$ to contain free variables that have the same value as some pseudo-free variable. In particular, in $message*$, the pseudo-free variable $m*_2$ (which is the mapping of m_3) is redudant to $m*_4$ and $m*_5$; any joins or projections that involve either $m*_4$ or $m*_5$ can use $m*_2$ instead because the recursive rule does not change $m*_2$. Similarly, the pseudo-free variable $m*_3$ (which is the mapping of m_4) is redundant to $m*_6$. Therefore, $m*_4$, $m*_5$ and $m*_6$ can be cut from $message*$ without data loss. We can similarly prune redundant attributes for $message**$. In general, pruning redudant attributes is a straightforward program analysis and rewrite post-processing procedure. After some variable renaming for clarity of exposition, the result is Listing 3.3.

```
1  % Prepare for transmission
2  message(@SourceA, SourceA, SinkB, DataC) :-
3    produce(@SourceA, DataC),
4    nexthop(@SourceA, SinkB, NextD).
5
6  % Route message to next hop parent until rendezvous (R1.1)
7  message(@NextE, SourceF, SinkG, DataH) :-
8    message(@CurrentI, SourceF, SinkG, DataH),
9    nexthop(@CurrentI, SinkG, NextE),
10   -rendezvous(@CurrentI, SinkG, DataH).
11
12 % Route interest back along next hop until rendezvous (R2,R3.1)
13 message*(@SinkJ, SinkJ, DataK, SinkJ, SinkJ, DataK) :-
14   interest(@SinkJ, DataK).
15 message*(@CurrentI, SinkG, DataH, SinkJ, SinkJ, DataK) :-
16   message*(@NextE, SinkG, DataH, SinkJ, SinkJ, DataK),
17   nexthop(@CurrentI, SinkG, NextE),
18   -rendezvous(@NextE, SinkG, DataH).
19
20 % At rendezvous, join message and interest and send to Sink (R4.2,R5,R6)
21 message**(@U1, U3, U4, SinkJ, SinkJ, DataK, U2) :-
22   message(@U1, U2, U3, U4),
23   message*(@U1, U3, U4, SinkJ, SinkJ, DataK),
24   rendezvous(@U1, U3, U4).
25 message**(@NextE, SinkG, DataH, SinkJ, SinkJ, DataK, U2) :-
26   message**(@CurrentI, SinkG, DataH, SinkJ, SinkJ, DataK, U2),
27   nexthop(@CurrentI, SinkG, NextE).
28 consume(@SinkJ, SinkJ, DataK, U2) :-
29   message**(@SinkJ, SinkJ, DataK, SinkJ, SinkJ, DataK, U2).
30
31 % What is consumed?
32 consume(@Sink, Data)?
```

Listing B.2: Rewritten `BasicProg` before pruning redundant attributes.

B.2 Session Rewrite

Listing B.3 and Listing B.4 show the result of applying Session Rewrite to Listing 3.4. Specifically, we show the execution according to Figure 3.6a. Also, variable names have been renamed and redundant attributes have been pruned for comprehensibility.

B.3 Routing Rewrite

Listing B.5 shows the result of applying Routing Rewrite to Listing 3.1. The rewritten program performs multiple instances of Routing Rewrite at once, because every forwarding node is simultaneously considered a "stateful server" with a proxy decision to be made. In other words, using Figure 3.7b as reference, *hop y* is just one "stateful server"; every forwarding node between *rendezvous* and *sink* is also a"stateful server."

In the general case, the optimizer can reassign any forwarding node's state. More typically, the optimizer considers path segments consisting of a sequence of forwarding nodes, and assigns the state for all nodes in the path segment to the same rendezvous. For example, in Figure 3.7b all forwarding nodes between *rendezvous* and *hop y* have their state reassigned to *rendezvous*. To specify this, Routing Rewrite employs a helper relation, $rendezvousIn$, to indicate the path segment that is to be source routed. Like *rendezvous*, *Decision Making* fills in entries for $rendezvousIn$ to correspond to its optimal decision making.

Listing B.5 perform the familiar *message* forwarding along *nexthop* until the *rendezvous* (lines 2-5). *nexthop* ships its data back toward the *rendezvous* via $nexthop*$ messages (lines 8-19). Note that $rendezvousIn$ limits the *nexthop* tuples to which this applies. Once $nexthop*$ and *message* rendezvous, $nexthop**$ tuples are created from $nexthop*$, and these serve as the source route for *message* as it traverses forward (lines 22-34).

```
1  % Client: Send request message to Proxy.
2  message(@Client, Client, Server, Request) :-
3    interest(@Client, Server, Request).
4  message(@Next, Client, Server, Request) :-
5    message(@Current, Client, Server, Request),
6    nexthop(@Current, Server, Next),
7    -rendezvous(@Current, Client, Server, Request).

9  % Server: Send session message to Proxy
10 message*(@Server, Server, Client, Data) :-
11   session(@Server, Client, Data),
12 message*(@Prev, Server, Client, Data) :-
13   message*(@Current, Server, Client, Data),
14   nexthop(@Prev, Client, Current),
15   -rendezvous(@Current, Client, Server, Request).

17 % Proxy: Combine request and session and send to Server
18 message**(@Current, Client, Server, Data, Request) :-
19   message(@Current, Client, Server, Request),
20   message*(@Current, Server, Client, Data),
21   rendezvous(@Current, Client, Server, Request).
22 message**(@Next, Client, Server, Data, Request) :-
23   message**(@Current, Client, Server, Data, Request),
24   nexthop(@Current, Server, Next).

26 % Server: Upon message, transition session state
27 session(@Server, Client, NewData) :-
28   message**(@Server, Client, Server, Data, Request),
29   transition(@Server, Data, Request, NewData).

31 % ... and respond to Client.
32 message(@Server, Server, Client, NewData) :-
33   message**(@Server, Client, Server, Data, Request),
34   session(@Server, Client, NewData).
```

Listing B.3: Rewritten `SessionProg`, a client-server roundtrip with session state proxy.

```
35 % Client: Consume response.
36 consume(@Client, Data) :-
37   message(@Client, Server, Client, Data),
38   interest(@Client, Server).

40 % Query: What is consumed?
41 consume(@Client, Data)?
```

Listing B.4: Rewritten `SessionProg`, a client-server roundtrip with session state proxy. (*Cont.*)

```
1  % Regular message routing until rendezvous
2  message(@Next, Src, Sink, Data) :-
3    message(@Curr, Src, Sink, Data),
4    nexthop(@Curr, Sink, Next), Curr != Sink,
5    -rendezvous(Next).
6
7  % Building source route: self-traversal backward
8  nexthop*(@Curr, Curr, Sink, Next) :-
9    rendezvousIn(@Curr),
10   -rendezvousIn(@Next),
11   nexthop(@Curr, Sink, Next).
12 nexthop*(@Curr, Curr, Sink, Next) :-
13   rendezvousIn(@Curr),
14   rendezvousIn(@Next),
15   nexthop(@Curr, Sink, Next),
16   nexthop*(Curr, Next, Sink, _).
17 nexthop*(@Prev, Link1, Sink, Link2) :-
18   nexthop(@Prev, Sink, Curr),
19   nexthop*(@Curr, Link1, Sink, Link2).
20
21 % Using source route: self-traversal forward
22 nexthop**(@Next, Link1, Sink, Link2) :-
23   nexthop*(@Curr, Link1, Sink, Link2), Curr != Sink,
24   message(@Curr, Src, Sink, Data),
25   nexthop(@Curr, Sink, Next).
26 nexthop**(Next, Link1, Sink, Link2) :-
27   nexthop**(Curr, Curr, Dest, Next),
28   message(Curr, Src, Dest, Data),
29   nexthop**(Curr, Link1, Sink, Link2), Link1 != Curr.
30 message(Next, Src, Sink, Data) :-
31   nexthop**(Curr, Curr, Dest, Next),
32   message(Curr, Src, Dest, Data),
33   nexthop**(Curr, Link1, Sink, Link2), Link1 != Curr,
34   -nexthop(Curr, Link1, Sink, Link2).
```

Listing B.5: Rewritten Routing Rewrite, a DVR-SR hybrid.