Title
Exploring a Centralized/Distributed Hybrid Routing Protocol for Low Power Wireless Networks and Large-Scale Datacenters

Author
Tavakoli, Arsalan

Publication Date
2009

Peer reviewed|Thesis/dissertation

Exploring a Centralized/Distributed Hybrid Routing Protocol for Low Power Wireless Networks and Large-Scale Datacenters

by

Arsalan Tavakoli

B.S. (University of Virginia) 2005
M.S. (University of California, Berkeley) 2008

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Scott Shenker, Chair
Professor Ion Stoica
Professor Steven Glaser

Fall 2009

Abstract

Exploring a Centralized/Distributed Hybrid Routing Protocol for Low Power Wireless Networks and Large-Scale Datacenters

by

Arsalan Tavakoli

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Scott Shenker, Chair

Large scale networking has always embraced distributed solutions. Centralized systems elicit knee-jerk reactions, typically pointing to a single-point of failure, difficulty maintaining global state, and operational latency. Nonetheless, centralized solutions have gradually begun to make headway in mainstream networks, such as enterprise networks. In this work, we take this trend one step farther, exploring centralized solutions to two extreme networking environments: Lossy and Low-Power Wireless Networks, and Large-Scale Datacenters.

Low-Power Wireless Networks can be characterized as dynamic high-churn environments with low-bandwidth radios. We present HYDRO, a hybrid routing protocol for low-power wireless networks. At its core, HYDRO forms a directed acyclic graph (DAG) that is locally maintained to support many-to-one collection based routing. In addition, topology reports from individual nodes are gathered to create a *sufficient* global topology view, which subsequently allows for centrally installed state in the network to optimize point-to-point communication.

Within the datacenter context, we focus on the difficulties of incorporating middlebox traversal requirements into the existing architecture. We begin by presenting PLayer, a policy-aware switching layer for datacenters that enables network administrators to explicitly dictate the middlebox traversal sequence of classes of traffic in their network. Given PLayer's predominantly distributed nature, we subsequently present a centralized PLayer design, discussing its ability to handle the demanding scalability requirements of datacen-

ters and provide a comparison of the two designs.

_____

Professor Scott Shenker
Dissertation Committee Chair

Dedication

To my loving parents, Sonbol and Amir, who have consistently shown me support and shepherded me in their own indirect ways, and my brother Arastoo, who continues to a be firm believer in criticism and mockery as the ultimate motivational means for success.

Contents

Dedication i

Contents iv

List of Figures v

List of Tables vi

Acknowledgements vii

1 Introduction 1
- 1.1 Routing in Networked Systems . . . 2
  - 1.1.1 The Internet At-Large . . . 2
  - 1.1.2 Low-Power Wireless Networks . . . 3
  - 1.1.3 Datacenters . . . 4
- 1.2 Shortcomings of Today's Solutions . . . 5
  - 1.2.1 Low Power Wireless Networks . . . 6
  - 1.2.2 Datacenters . . . 7
- 1.3 Distributed vs. Centralized . . . 7
  - 1.3.1 A Hybrid Compromise . . . 9
- 1.4 Contributions . . . 10
  - 1.4.1 HYDRO: A Hybrid Routing Protocol for Low Power Networks . . . 10
  - 1.4.2 PLayer: A Policy-Aware Switching Layer for Datacenters . . . 11
- 1.5 Thesis Roadmap . . . 11

2 Background / Related Work for Lossy and Low-Powered Networks 13
- 2.1 Lossy and Low Power Networks (L2N) . . . 13
  - 2.1.1 Constrained Resources . . . 14
  - 2.1.2 Typical Applications and Usage . . . 14
- 2.2 Routing Over Lossy and Low Power Networks (ROLL) . . . 15
  - 2.2.1 Core Challenges . . . 16
  - 2.2.2 ROLL Protocol Requirements . . . 18
- 2.3 Difficulties in deploying Internet-based solutions for ROLL . . . 20
  - 2.3.1 Stable Backchannel To Centralized Controller . . . 20
  - 2.3.2 Consistent Global View of Dynamic Topology . . . 21

2.3.3 Providing Reliability Over Lossy Links . . . 22
2.4 Existing Low-Power Protocols . . . 22
2.4.1 Collection/Dissemination Oriented Protocols . . . 23
2.4.2 Other Routing Protocols . . . 24
2.5 Internet Routing Protocols . . . 26
2.5.1 Centralized Routing Architectures / Protocols . . . 26
2.5.2 Mobile Ad-Hoc Networking (MANET) Protocols . . . 31
2.6 Summary . . . 32

3 HYDRO: A Two-Tiered Hybrid Routing Protocol for L2Ns 33
3.1 Design Overview . . . 33
3.1.1 Default Route Establishment . . . 35
3.1.2 Global Topology View . . . 36
3.1.3 Centralized Route Install . . . 38
3.2 Our Hybrid Approach . . . 41
3.2.1 Controller Design . . . 41
3.2.2 Shortcomings of Centralized Protocols . . . 43
3.2.3 ROLL Requirements . . . 44
3.3 Extensions and Future Work . . . 45
3.3.1 Multiple Controllers . . . 45
3.3.2 Multicast . . . 46
3.3.3 Policy-Based Routing . . . 47
3.4 Design Limitations . . . 47
3.5 Summary . . . 49

4 HYDRO Evaluation 50
4.1 Evaluation Methodology . . . 50
4.2 Testbed Evaluation . . . 53
4.3 MATLAB Simulations . . . 54
4.4 TOSSIM Evaluation . . . 59
4.5 State and Control Overhead Analysis . . . 64
4.5.1 State . . . 64
4.5.2 Control Overhead . . . 65
4.5.3 Analysis . . . 66
4.6 Revisiting the Core Challenges . . . 67
4.6.1 Routing/Transmission Stretch . . . 67
4.6.2 Routing State . . . 67
4.6.3 Network Overhead . . . 68
4.7 Summary . . . 68

5 PLayer: A Policy-Aware Switching Layer for Data Centers 69
5.1 Datacenter Routing and Infrastructure . . . 69
5.1.1 Datacenter Network Architecture . . . 70
5.1.2 Limitations of Existing Mechanisms . . . 70
5.2 PLayer Design Overview . . . 75

5.2.1 PLayer Goals . . . 75
5.2.2 Policy-Aware Switches . . . 76
5.2.3 Policy Specification . . . 76
5.2.4 Centralized Components . . . 77
5.3 PLayer Performance . . . 77
5.3.1 Implementation . . . 77
5.3.2 Preliminary Evaluation Results . . . 78
5.4 Summary . . . 79

6 Centralizing PLayer **81**
6.1 NOX: A Centralized Control Platform for Networking . . . 81
6.1.1 Architectural Overview . . . 82
6.1.2 OpenFlow Programmable Software Switches . . . 83
6.1.3 NOX Centralized Controller . . . 84
6.2 Centralized PLayer Implementation Over NOX . . . 85
6.2.1 Design Overview . . . 85
6.2.2 Initial Evaluation Results . . . 88
6.2.3 Extensions and Future Work . . . 89
6.3 Comparison of Both Implementation Paradigms . . . 91
6.4 Summary . . . 92

7 Conclusion **93**
7.1 Contribution Summary . . . 93
7.2 Analysis and Discussion of Future Roadmap . . . 94

Bibliography **105**

List of Figures

3.1 An example collected topology when only one link is reported by each node. By default we report four links per node. 37
3.2 The effect of the route install primitive. 39

4.1 79-node Smote Testbed at UC Berkeley 51
4.2 Smote Testbed (50 nodes) - bidirectional ping flows 54
4.3 25 nodes of the Smote testbed at -22dBm 55
4.4 100 Node Uniformly Distributed Network 56
4.5 225 Nodes Arranged in a 15 x 15 Grid 58
4.6 225 Nodes Arranged in a 15 x 15 Grid with Local Destinations 58
4.7 Our default route selection protocol compared with CTP in simulation with a poor noise model, and to results on our testbed. 60
4.8 Simulation results from 225 Nodes arranged in a 15×15 Grid 61
4.9 225 Nodes Arranged in a 15 x 15 Grid - 5 Nodes failing every two minutes, with 10 concurrent flows 63

5.1 (a) Prevalent 3-layer datacenter network topology. (b) Layer-2 path between servers *S1* and *S2* including a firewall. 71
5.2 Topologies used in benchmarking *pswitch* performance. 79

6.1 Example NOX Network Architecture 82
6.2 An example of an ambiguous previous hop during policy traversal, assuming middleboxes are to be traversed in numeric order. 88

List of Tables

3.1 Key design questions and our decisions 40

4.1 HYDRO State Requirements 64
4.2 HYDRO Control Overhead 66
4.3 Total Costs of HYDRO and S4 (Per Node) 67

Acknowledgements

Acknowledgements are always difficult because of the fear that a momentary slip-of-the-mind will lead to exclusion of people who really do deserve credit and recognition.

At the top, a great deal of thanks and gratitude must be directed towards my advisor, Professor Scott Shenker, who understood from the beginning that the best way to interact with me was to walk a fine line between taking a hands-off approach to allow me to find my own way, while at the same time being there to provide high-level feedback and bluntly letting me know when certain ideas were dead-ends.

I must also thank my student collaborators that made the specific research discussed in this dissertation possible. Stephen Dawson-Haggerty and I spent months debating the various dimensions of HYDRO, and our often opposing views helped foster a more thorough and balanced exploration of our design decisions. In addition, Steve's implementation of the underlying IP stack and significant portions of HYDRO made this research possible. Switching to PLayer, I thank Dilip for allowing me to get involved with his research, and the hours of discussions we had that would form the basis for much of my later work in datacenters and networking in general.

Although I only had a single advisor, three other professors in particular were invaluable to me during my PhD tenure. Professor David Culler was always willing to take the time to push me on my research, probing to find the inevitable holes that would always make the work stronger down the line. Professor Ion Stoica consistently reminded me that while a high-level architectural view was needed, attention to the smaller details should never

be sacrificed. And last, but not least, Professor Phil Levis, who despite being swamped with trying to get his own academic career off and running, consistently took the time to proactively check-in and discuss the latest updates to my research and provide his input.

Finally, I have to thank all of the occupants of Soda 410. Much of my research stemmed from ad-hoc discussions that started with lab-mates, and beyond research, they provided an a social-environment that made it an enjoyable place to do research, without which my productivity would have suffered immensely, oddly enough. I thank you all.

Chapter 1

Introduction

Routing, by one definition [9], is *the process of selecting paths in a network along which to send network traffic*. Looked at from another perspective, it can be divided into a three step process: 1) Gathering information about the state of the network, 2) Inputing this information into an objective function that optimizes across a set of metrics, and 3) Executing the selected routing decision. Numerous potential combinations exists, resulting from a wide range of mechanisms in the design space to address each of the three steps. Network routing research is not a new phenomenon (and routing, for that matter, is a fundamental challenge in many non-data based fields as well), yet growth in size, complexity, and heterogeneity have consistently provided new hurdles and challenges that must be overcome.

One of the principle design considerations in routing has been the fundamental tension between centralized and distributed mechanisms. In regards to the process identified above, these two design paradigms have differing philosophies on the co-location and degree of intertwining of the three steps of the routing process. Centralized mechanisms provide a single point of control, and facilitate understanding of the operation of the entire system; however, they also require global network state, which can be significant, and suffer latency costs in responding to local phenomena. Distributed mechanisms rectify the latter two concerns, and alleviate scalability concerns, but are also notoriously difficult to debug and

diagnose. Both of these mechanisms have been explored in previous routing research.

This dissertation focuses on synthesizing the key tradeoffs between centralized and distributed mechanisms in two extreme cases of networking: high-churn, low-bandwidth environments at one end of spectrum, and low-churn, high-bandwidth environments at the other end. In this chapter we begin by briefly describing how routing is performed in a variety of different network environments, subsequently highlighting the shortcomings of today's solutions, particularly in the low-power wireless network domain. We then explore in additional detail the merits of distributed and centralized routing, and propose the need for a hybrid compromise. Last, we highlight our two main contributions, a hybrid routing protocol for low-power networks, and the centralizing of a policy-drive switching layer for datacenters, and conclude by laying out the roadmap for this dissertation.

1.1 Routing in Networked Systems

In exploring routing in modern day systems, we begin by examining the general umbrella structure of the Internet At-Large, and then examine our two target environments: Lossy and Low-Power Networks (L2Ns), and Large-scale Datacenters.

1.1.1 The Internet At-Large

The Internet is a global network of interconnected computers, and its complex physical and administrative structure make routing across end-points a challenging task. Roughly, it is formed by a set of Autonomous Systems, or AS's, which are administrative domains under the control of (typically) a single network operator that present a unified routing policy to the rest of the Internet. The set of applications and services that operate over the Internet is vast, leading to a wide variety of necessary traffic paradigms and network usage. Two forms of routing exist: inter-domain routing, which involves communicating end points that are in different AS's, and hence require coordination, and intra-domain routing, which

is restricted to communication contained within a single AS.

This dissertation focuses on specific instances of intra-domain routing, as the lack of a single uniform routing policy makes centralized routing untenable in the inter-domain context.

1.1.2 Low-Power Wireless Networks

Lossy low-power wireless networks (L2Ns) are composed of a subnet of embedded networking devices (Node Routers, or Nodes), with a modest number (relative to the size of the subnet) of routers, gateways, or bridges (Border Routers) providing ingress/egress connectivity to the backbone network. The in-network nodes are resource-starved along numerous dimensions (e.g. storage, energy, bandwidth, processing-power), and subject to dynamic variation in link quality due to environmental and other factors. The challenges of serving this class of networks with conventional routing protocols have resulted in an IETF draft that identifies five quantifiable criteria that are necessary for L2N routing protocols [44]:

- **Table Scalability:** Each node's state must be bound by the number of unique destinations it communicates with.
- **Loss Response:** Any response to loss is localized to the affected data flow.
- **Control Cost:** Control overhead must be bound by the periodic data traffic rate.
- **Link Quality:** Protocol must define a metric for differentiating among links.
- **Node Cost:** Algorithm must have some notion of heterogeneity in the network.

While the last two criteria mandate an understanding of heterogeneous wireless networks, the first three criteria highlight the fundamental tension between the optimality of the routes and the cost of discovering and maintaining the route. We define stretch as the

ratio between actual path length formed by the protocol and the optimal shortest path, and use this as a proxy for route efficiency.

The main strength of L2Ns lie in their ability to be easily deployed (because of their small size and wireless communication abilities), and their extended lifetimes (because of low duty cycles and potential for solar energy sources in battery-powered deployments). Coupled with their ability to support a wide variety of sensors, L2Ns are predominantly used in monitoring and actuating application domains, such as industrial monitoring [59], building automation [53], and smart homes [14]. These networks can be quite large, such as deployments with thousands of Nodes, and deep, with some application domain routing requirement documents listing the potential for networks that are 20 hops deep [59].

Traffic patterns in L2Ns are atypical relative to other networks. One of the most important traffic patterns is all-to-one collection paradigms (and the reverse one-to-all dissemination paradigm), and these must be highly optimized. Unicast flows are critical as well, but the number of unicast destinations in the network is much smaller than the total number of nodes. Finally, to conserve energy in many applications, data rates are very low, and the nodes themselves are potentially only awake 1% of the time or less.

Any routing protocol that is to be effective must operate within these confines, while maintaining the five IETF properties highlighted previously.

1.1.3 Datacenters

Datacenters are densely populated large-scale computer warehouses, typically designed to provide support for web-based or networked applications. They are characterized by high-bandwidth low-latency links, which allow for powerful and responsive distributed applications. Single enterprises, such as financial corporations, can use them to provide the necessary storage and computational horsepower in a highly responsive and available manner to employees, or conversely, Datacenters can also be used to provide global web applications, such as web-mail and online file storage.

The challenge with datacenters is the complexity of the necessary routing policies. Datacenters often make heavy use of *middleboxes*, which are specific capability devices, such as firewalls, load balancers, and SSL offload boxes, that must be traversed by packets before arriving at the requested end-host. Datacenters are often characterized by very heavy loads, making software-based slowdowns costly, and often multiple instances of these middleboxes are necessary to support all applications and services. Given the potential failures of links and nodes, a graceful degradation process is needed while those failures are being rectified. Finally, efficiency dictates that middleboxes and resources should be shared across policies, even those that have conflicting policy specifications (e.g. the traversal orders of the same middleboxes do not match).

Traditional methods have involved ensuring that these boxes are on the physical path, which requires spanning tree algorithm constraints and engineering, or reliance on other methods such as VLANs. As pointed out in [42], such techniques are hard to maintain, cumbersome to update in the face of topology changes, and unable to provide correctness guarantees in the face of certain types of link and node failures. What is essentially needed is the ability to separate logical (policy-oriented) and physical (link-based) topologies, and there are existing systems that aim to provide this ability, with varying degrees of success. We discuss the shortcomings of existing solutions in 1.2.

1.2 Shortcomings of Today's Solutions

Routing in each of our two domains has received significant research exposure, yet serious shortcomings remain in each. We briefly highlight these shortcomings here, and examine them in more detail in chapters 2 and 5.

1.2.1 Low Power Wireless Networks

L2N specific routing protocols exist, albeit mostly at layer 2, with S4 [52] providing the most compelling state-stretch tradeoff. S4, a compact-routing based protocol, guarantees a stretch upper bound of 3, while maintaining $O(\sqrt{N})$ state on average at each node, where N is the number of nodes in the network. This alone fails the first of the five IETF criteria. In addition, landmark-based routing schemes, such as S4, require out-of-band landmark selection and group maintenance mechanisms, which are difficult to provide within the strict confines of L2Ns. Nonetheless, we compare our solution to S4 with respect to state, stretch, and reliability in chapter 4.

A host of existing routing protocols are available in other domains, and we only discuss them here briefly; Chapter 4 provides a more thorough analysis. DSR, AODV, OLSR, and DYMO are examples of prevalent MANET (Mobile Ad-Hoc Networking) ad-hoc wireless network protocols. These protocols are designed for wireless networks with resource-abundant mobile nodes, where communication between all nodes is equally likely and must be supported. This results in large routing state and/or significant control traffic, which is impractical in our target environment. In fact, these protocols, and other existing wired and wireless protocols are evaluated against the previously mentioned five criteria in [44], and the survey concludes that none of the protocols (in their current form) satisfy all five requirements (or even four out of the five).

We also observe that existing protocols for L2Ns are predominantly distributed solutions. Meanwhile, centralized solutions for large, specialized wired networks (e.g. data-centers and enterprise networks), such as MPLS [62] and Ethane [17], have successfully managed state and stretch concurrently. However, the high-bandwidth, low-churn nature of these environments was critical to the success of these solutions, while L2N networks are inversely characterized by low-bandwidth and high churn. Many of the implicit assumptions of these high-end networking environments prevent a natural design of a centralized protocol for L2Ns.

1.2.2 Datacenters

Routing solutions for large-scale enterprise networks often claim natural applicability for datacenters as well, yet this fails to heed the notion that middleboxes play a large role in these networks, and that the large number of applications and hosts supported by these networks mandate solutions that encourage sharing but eliminate waste of processing resources.

Many current solutions primarily rely on distributed mechanisms, in an attempt to simply port existing layer-2 switching solutions to the datacenter, often conflating the logical and physical topology of the network. The sheer scale of these networks and the complexities of the interwoven routing policy specifications makes such distributed mechanisms extremely difficult to implement and maintain. In addition, correctness guarantees are difficult to provide in such distributed solutions. Conversely, the high throughput needed and significant load on the network make the latency and overhead of purely centralized algorithms costly and often impractical.

Centralized solutions, such as Tesseract [72] and 4D [32], do exist, providing a centralized control plane. However, the focus of these systems is often to build the data plane for point-to-point communication, rather than explicitly addressing the need to traverse middleboxes along a given path. Conversely, systems such as i3 [66] and DOA [69] enable indirection and explict traversal of middleboxes along a path, typically dictated by the route destination. However, these mechanisms are designed for internet-scale routing, and hence are too costly and inflexible for datacenter operation.

1.3 Distributed vs. Centralized

Distributed algorithms typically allow nodes to make decisions locally, forgoing the need to communicate with a central entity. Each node gathers all needed information, and bases routing decisions on this information. One key aspect of distributed algorithms is their

fault tolerance and scalability. Since nodes gather and process information locally, the affect of failures of other nodes is mitigated, and new nodes can join the network and begin executing the distributed algorithm immediately. Distributed algorithms also provide local agility, in that a node can overcome failures and changes in the topology by recomputing its routing objective function locally given the new information.

The shortcomings of distributed algorithms are complexity, and inconsistency. Understanding all the interactions between the distributed components of a large-scale complex system is a daunting task, particularly when attempting to diagnose erratic or undesirable behavior. A distributed algorithm, by definition, takes a single concept and breaks it up into smaller pieces that can be executed in parallel, pushing all complexity, state, and processing to the individual nodes. Recreating the notion of a single aggregate *network* in such scenarios is challenging. Many times the cause of this erratic behavior stems from inconsistent state across the nodes, which is often times difficult to detect, and even more difficult to replicate due to the nondeterminism of race conditions in real environments. Mechanisms such as full logging [28] and replay [27] exist for diagnosing such behavior, but even these become difficult to use as the *correctness predicates* become increasingly complicated.

Centralized routing algorithms provide a clean interface for policy specification. Global network state, including topology, is obtained from network nodes and maintained at a centralized controller. The goal is to push complexity and state to the edges of the network, and task the in-network nodes with only simplified execution of routing decisions, which are communicated to them by the controller. Minimizing the distributed aspects of the algorithm allows for easier understanding of the routing operations, and subsequently more complex routing policies.

Centralized algorithms have their disadvantages as well. The presence of a centralized controller indicates a single point of failure. This controller can remain a single logical entity but have multiple physical replicas, yet this still doesn't prevent the formation of traffic bottlenecks and congestion. The need to create and maintain a global view of the topology

also results in a significant amount of network control traffic being generated by the nodes and sent to the controller continuously. A bootstrapping process is also necessary to create a reliable control plane that functions independently of the data plane. Finally, there is an inherent lag and overhead to centralized routing algorithms. Nodes must incur the cost and latency of transmitting network state to a controller, and waiting for the requested routing decision for every new network event, preventing any notion of local agility.

1.3.1 A Hybrid Compromise

Most routing paradigms embrace the extreme (or near extreme) points of the design spectrum: either fully centralized or fully distributed. However, this dissertation postulates that in the context of Lossy and Low Power Wireless Networks and Large-Scale Datacenters, a hybrid routing protocol that embraces centralized control and local agility enables the network to reap the benefits of both design styles.

For simplification, we can divide routing decisions into two categories: *major*, and *minor*. While not a crisp separation, we define a *major* decision as a path-level decision, such as discovering a route to any arbitrary destination, and a *minor* decision to be one that simply involves choosing among various next-hop links. We propose to let centralized components dictate the *major* decisions, but task the in-network nodes with the *minor* decisions.

As an example, the centralized controller can use the global network view to discover the top 3 paths for routing between two nodes, A and B, in the network, according to a high-level routing policy. It subsequently provides all three options to the appropriate nodes in the network. When a packet needs to be routed from A to B, A is tasked with deciding which of the three routes to use according to locally observed network conditions. Such a policy allows the centralized controller to utilize its view of the network, and prevent an expensive distributed discovery operation, while also allowing nodes to adapt to locally changing conditions without needing to incur the cost of communicating with the

controller.

1.4 Contributions

The contributions of this dissertation are two-fold, with one for Lossy and Low Power Networks, and the other for routing in Datacenters:

1.4.1 HYDRO: A Hybrid Routing Protocol for Low Power Networks

The success of centralized routing solutions for large-scale networks is well documented, yet many of the factors which contribute to this success are non-existent in L2Ns, mainly due to the resource-starved nature of the nodes and the vagaries of low-power wireless networking which create a dynamic environment even with stationary networks. By the same token, distributed protocols have struggled to provide an effective solution that works efficiently while meeting the minimal state, overhead, and energy requirements.

We introduce HYDRO, a hybrid routing protocol that leverages the inherent two-tiered structure of L2Ns. Using local information, nodes in the network discover and maintain multiple potential routes to a centralized controller, providing the needed control plane connectivity. Based on periodic topology reports received from the nodes, the controller builds a global view of the topology, and can provide an efficient path between any two nodes based on a specified routing policy. When a node wants to route to a particular destination, it uses a route provided by the controller, and if no applicable one exists, it routes the packet to the controller using a locally maintained path and simultaneously requests a path for subsequent packets.

We provide an overview of the design of HYDRO, and also evaluate its performance, both in simulations and a real-world testbed, demonstrating that it provides significant improvements over existing solutions.

1.4.2 PLayer: A Policy-Aware Switching Layer for Datacenters

We provide a brief overview of PLayer, a Policy-Aware Switching Layer for Datacenters, which allows for explicit policy specification, and provides guaranteed policy adherence, even in the face of link, node, and policy churn. PLayer was designed in a predominantly distributed fashion in which full policy specifications are pushed to all switches, eliminating the cost of communicating with a centralized controller. As such, while it enjoys the benefits of a distributed approach, it is also limited by its shortcomings.

In an effort to provide a fair comparison, we implement the high-level design of PLayer over NOX, a centralized open-source platform for writing network management software, providing a purely centralized point in the design space. Subsequently, we compare and contrast these two versions across various metrics, and also exampine the applicability of the *hybrid compromise* paradigm discussed previously.

1.5 Thesis Roadmap

This dissertation is organized as follows: We begin in chapter 2 by examining existing literature within the low power wireless space, highlighting work that has provided us with invaluable building blocks. In addition, we motivate our work by describing the needs of applications in the space and the shortcomings of existing solutions, while also presenting metrics and requirements for evaluating solutions. In chapter 3 we present HYDRO, our hybrid routing protocol. The chapter provides a comprehensive examination of the challenges of building a centralized solution for L2N networks, the design decisions that led to our final design, and the pros and cons of each of them. We present our evaluation of HYDRO in chapter 4 across a host of experimental environments, both real-world and analytical. In chapter 5, we turn our attention to PLayer, discussing datacenter networking in general and the motivation for our solution, as well as its design and evaluation. Subsequently we focus on centralizing PLayer in chapter 6, outlining the challenges and comparing it with the

original PLayer design. Finally, we conclude in chapter 7 by discussing the implications of our work and the future directions for our research.

Chapter 2

Background / Related Work for Lossy and Low-Powered Networks

In this section, we explore the constraints of routing in low power wireless networks, and examine existing solutions in the space, highlighting their shortcomings and also elements which serve as building blocks for our design. Also, we readily acknowledge that creating a distinct control plane for routing is not a novel idea, and examine related works in general networking. Many such works served as inspiration for our design, yet we also discuss barriers that prevent such works from being naturally portable to our class of networks.

2.1 Lossy and Low Power Networks (L2N)

Lossy and Low Power Networks were designed to easily integrate into their environment and provide a wealth of information about their surroundings. Each device typically contains a set of sensors, with a wide variety of options (e.g. temperature, humidity, light, magnetometer, and accelerometer), as well as a low-power wireless radio.

2.1.1 Constrained Resources

One defining characteristic of L2Ns is their resource-starved nature. The most common platform used today provides 48KB of code space, and 10KB of memory, coupled with a 256Kbps low power radio [61]. The memory aspect provides a key limitation in the networking context, as it provides tight bounds on the state that can be maintained at each node.

Energy is also a principle concern with such nodes. In many deployments, the nodes are powered by batteries, and more recently, solar harvesting as well [38]. Given the desire for long term deployments (perpetual if possible), the devices' energy budgets are often limited to sub-1% duty cycles (as energy source replacement, i.e. changing batteries, is typically not practical), making energy conservation a key design principle. Even in situations where a power source exists, energy efficiency still remains a key goal. Given that the most expensive operation is radio communication (both sending and receiving), many of the routing constraints we impose in section 2.2 emerge naturally.

2.1.2 Typical Applications and Usage

L2Ns have been used in a wide variety of scenarios. The most typical use involves monitoring, whether environmental or industrial. Without the need for existing infrastructure, these devices can be more easily deployed in their target environments, and the wireless capabilities and long-term lifetime allow for consistent data gathering.

In addition to monitoring, potential applications include building automation [53] and smart homes [14], elderly patient assistance [18], volcano monitoring [70], and energy metering [36].

With such applications, the most natural communication primitive is collection, or many-to-one routing. The number of data sinks can be increased for robustness and scalability, but the number of destinations is far less than the number of data sources. Any routing protocol should not only provide effective performance for this traffic paradigm,

but also leverage it to provide additional services.

Other traffic patterns exist as well. Multicast traffic is used, either when using group management applications, or for communicating with localized geographic clusters. Unicast flows are the primary vehicle for diagnostic tools and applications, as well as actuating systems. One key characteristic is that in the majority of cases, the number of destinations a node communicates with is a very small subset of the entire network.

The final note about traffic patterns is that data rates tend to be very low, mainly to conserve energy. In many scenarios, this means that a single piece of data is communicated every hour or so. Given the need to conserve energy, control traffic and network overhead must be proportional to this data rate, which we address below.

2.2 Routing Over Lossy and Low Power Networks (ROLL)

Routing for L2Ns is a critical component of the functionality necessary for effective deployments. Many early deployments were hampered with difficulties routing the data collected back to a base station [68, 50]. These systems have certainly progressed since that point, yet the community has struggled to build an open routing protocol that moves beyond simple collection and dissemination communication primitives (although some proprietary solutions do exist).

In the first part of this section, we examine the three core challenges that need to be overcome when developing a practical routing protocol for L2Ns: minimizing transmission stretch, routing state, and network overhead.

In order to tackle this problem of building a routing protocol for L2Ns, the ROLL (Routing over Lossy and Low Power Networks) Working Group was formed within the IETF. Within this group, application requirement documents were published specifying the routing requirements for four widely deployed application domains: Industrial Monitoring [59], Building Automation [53], Urban Environments [20], and Home Automation [14]. Sub-

sequently, as part of a protocol survey [48], the intersection of these requirements (i.e. a set of necessary but not necessarily sufficient requirements for any ROLL routing protocol) was taken and synthesized into five quantifiable metrics, which we discuss below.

In chapter 4, we return to these challenges and metrics and determine how well our proposed routing protocol fares.

2.2.1 Core Challenges

There are three specific challenges that arise in the context of routing in L2Ns, all of which stem from scarce energy and memory resources. While work in other domains considers these elements as well, they are typically much more loosely bounded, and often an optimization, rather than a necessary operating condition. The overarching challenge in designing a protocol for ROLL that addresses all these challenges is that in many typical systems, addressing one challenge is done at the expense of one or both of the other two.

Routing/Transmission Stretch

Routing stretch refers to the ratio of the length of a selected path (in terms of hops), to the length of the optimal path. It is meant to serve as a measure for an algorithm's path selection efficiency. In wired networks, with reliable links and stable topology, calculating the optimal path according to some policy (typically shortest-path) is straight-forward, as the shortest path is either known a priori, or can be calculated by taking a snapshot of the physical topology of the network.

In L2Ns, there is no clear *optimal path* for a multitude of reasons. First of all, links are inherently lossy, and consequently treating a link as a binary value is not useful. Furthermore, the vagaries of low-power wireless communication often create dynamically shifting connectivity graphs, again making it difficult to pin down a single route.

Rather than routing stretch, the ratio that makes more sense in ROLL environments is transmission stretch, which is bounded below by the routing stretch. Since links are lossy,

expected transmissions, or *ETX* can be used to determine the *cost* of a link. The subsequent calculation of an *optimal* path, which runs into the same problems as discussed above for routing stretch, can be based on a previous snapshot of the state of the network as a rough approximation, or by using another protocol's choices as the lower bound. As a side note, if non-direct paths are taken (such as is the case with opportunistic forwarding or in loss response scenarios), all transmissions associated with the packet must be accounted for in the cost of a given path.

Ultimately, the importance of this metric stems from the need for energy conservation. Radio usage, both for transmissions and receptions, is the most expensive operation for these low power devices, which is critical in networks where sub 1% duty cycles are required for long-term operation. Minimzing this stretch is particularly challenging as typical methods involve keeping global topology views at all routers and switches, kept consistent by the continuous flow of periodic traffic, neither of which is feasible in L2Ns.

Routing State

As discussed previously, the most common low-power platforms provide about 48KB of code space, and 10KB of memory, requiring minimal footprint by all system components, including the routing layer. Subsequently, routing state, including neighbor tables, forwarding tables, and routing information must be minimal. This is further complicated by the dense and large-scale nature of many L2N deployments.

Network Overhead

L2N deployments are often characterized by extremely low data rates, often in the range of a single message per hour or less, with duty-cycles of sub 1% not uncommon. Given that these uses stem from the need to minimze use of the radio, significant network overhead in the form of control traffic is infeasible. This includes consistent flooding of updates and topology information, as well control traffic directed at recovering from path breakage.

2.2.2 ROLL Protocol Requirements

The ROLL protocol survey [48] provides 5 quantifiable metrics that any routing protocol must satisfy in order to be considered for any of the four application domains discussed above.

To simplify the specification of the quantifiable metrics (in big-O notation), we provide the following formal definitions (from the protocol survey [48]):

- **N:** Number of nodes in the network.
- **D:** Number of unique destinations in the network.
- **L:** Size of a node's local, single-hop neighborhood (network density).
- **R:** Application data rate.
- ϵ**:** A small constant.

Table Scalability

This metric arises from the need for network scalability, coupled with the presence of memory scarcity. L2N deployments can potentially be composed of tens of thousands to millions of nodes [20], making it impossible for a node to store state about the entire network, and also unnecessary when the number of destinations, D, is typically much smaller than N. By the same token, node densities, L, vary widely between deployments, and can be significant in certain applications, prohibiting the storing of the entire single-hop adjacency table. Consequently, the size of the routing table must scale by the number of destinations in the network, $O(D)$.

Loss Response

The dynamic nature of L2Ns results in a significant amount of link churn, even relative to other wireless networking technologies. Flooding of global updates throughout the network

to update topology information is infeasible. Rather, loss responses must be limited to either the set of destinations actively using the link, *O(D)*, or a single local broadcast, *O(1)*, to notify a node's single-hop neighborhood.

Control Cost

Routing protocols require control traffic to discover the topology of the network, communicate routing table information, and maintain up-to-date routing state. However, given the low data rates typical in L2N deployments, and the requirement that energy consumption (and by association network overhead) be minimized, periodic or significant network traffic could lead to large network overhead on a per-data-packet basis. As such, the control cost must be bounded by the data rate, *O(R log(L))*. The *log(L)* results from retransmission requirements for reliability in L2Ns, and is further explained in the protocol survey [48].

Of course, in cases in which the data rate is very low, such as in some event-driven applications, a certain small amount of control traffic must be allowed for bootstrapping maintenance and discovery, resulting in a control cost bound of $O(R \log(L) + \epsilon)$, where ϵ is a small constant. Although this is not technically correct big-O notation, it was specified as such to make the allowance of the ϵ control traffic explicit.

Link Cost

Link quality varies greatly in L2Ns, and so a suitable routing protocol must take into account links characteristics as part of an objective function for varying metrics, such as limiting the number of transmissions or minimizing latency.

Node Cost

In addition to varying links, L2Ns are also composed of heterogeneous devices. The heterogeneity may stem from hardware characteristics, such as increased memory and networking bandwidth, or energy. This heterogeneity may also stem from a node's state, such

as the amount of energy remaining and routing load. The routing protocol must allow for the specification of these attributes and input them into the objective function that selects routing paths.

2.3 Difficulties in deploying Internet-based solutions for ROLL

The concept of centralized routing, or separating the control and data plane, is not a novel one; we explore several existing protocols later in this chapter that embrace the same centralized paradigm. However, these solutions were designed primarily for Internet based systems, which are characterized as low-churn, high-bandwidth environments. This is in stark contrast to the high-churn, low-bandwidth nature of typical L2N environments. Consequently, we discuss three factors that are critical to the success of centralized solutions, and while trivial to achieve in general networking environments, present the main obstacles to a centralized solution for ROLL.

2.3.1 Stable Backchannel To Centralized Controller

A logically centralized controller (there can be multiple physical controllers that act as a unified logical entity) serves as the brain of the network, gathering information from all the routers in the network, and also disseminating control commands. One of the key implications of this two-tier hierarchy is the need for a stable channel between the controller and all routers in the network at all times. In many cases, this channel uses logically (and in extreme cases physically) different links than the data plane, as its reliability and stability must be ensured.

In large wired networks, creation of this channel is simplified by underlying data link layers. In a predominantly IP and Ethernet based world, the underlying layer-2 spanning tree algorithm provides reachability between all switches and end-hosts in a subnet, and

layer-3 lookups can be accomplished using ARP, if needed. This is feasible because of the local broadcast nature of communications on a subnet. Extending this reachability across multiple subnets can be done by introducing layer-3 routing elements as well. Furthermore, once this channel has been established, it remains relatively stable, except in cases of severe congestion or physical partitioning, which are outside of the scope of our problem.

In contrast, no such natural mechanisms exist in L2N environments. The link-layer provides only single-radio-hop communications, with no inherent mechanisms for building neighbor tables. Networks can be relatively deep, with diameters of 20 or more hops possible [59]. Furthermore, the quality of links vary widely, both spatially and temporally, creating a lossy path that is highly dynamic (even with no physical mobility). Finally, this channel must be maintained with minimal control traffic and network overhead, per the requirements outlined previously.

2.3.2 Consistent Global View of Dynamic Topology

One of benefits of centralized routing is the ability to make routing decisions at a central location that has complete information about the state of the entire network. In order to create such a global view, all the elements of the network must register upon joining, and then provide updates of new neighbors and changes in the state. In many existing solutions, there are periodic heartbeats that are sent in the form of link state advertisements, which allows the controller to maintain this global view.

Three main factors complicate this task in L2Ns: control traffic restrictions, memory constraints, and dynamic topology. Control traffic must be bounded by the data rate, and so consistent periodic link state advertisements with updates sent to the controller are not practical. Furthermore, the loss-response ROLL criteria dictates that control traffic in response to a loss must be limited to active paths or the single-hop broadcast domain, preventing triggered global updates. Also, links undergo significant temporal and spatial variability. Links are not binary, and maintaining consistent views of network link qualities would re-

quire significant amounts of control traffic. Finally, memory constraints prevent individual nodes from maintaining state for all nodes in the network, or even the single-hop broadcast neighborhood, leading to the unavailability of needed information, even if control traffic was not problematic.

2.3.3 Providing Reliability Over Lossy Links

The target communication environment for the majority of existing routing protocols is often highly-reliable, in the form of either wired links, or single-hop wireless links that provide relatively high reliability, through both link-layer and typically transport layer retransmissions.

In L2Ns, the low-power nature of the radio leads to a small Signal-to-Interference-and-Noise Ratio (SINR) on many links. As demonstrated by Srinivasan et al. [63], the SINR of these links is often on the cusp of the necessary threshold to receive a packet, which results in bimodal links due to variations in signal strength and interference levels. Consequently, accurate link estimation is critical, both in accurately calculating a link's quality/cost, and also in identifying and accounting for temporal shifts in these link qualities in order to minimize transmission stretch while maintaining reliability.

2.4 Existing Low-Power Protocols

Routing has been an integral part of L2Ns since their initial inception, as data and commands had to be relayed between nodes. Initially, as many of the deployments focused on various types of monitoring, such as environmental, the predominant communication paradigm was *Collection-Oriented*, or *many/all-to-one* routing. By association *Dissemination-Oriented*, or *one-to-many/all*, routing also emerged because of the need to send commands to the nodes (e.g. for time synchronization). We examine the progression and state-of-the art for these constrained L2N routing protocols.

In addition, as the set of potential applications domains expanded, the need for richer and fuller routing protocols became clear, i.e. those that would support unicast and multicast communication paradigms as well. We also examine these in this section as well.

2.4.1 Collection/Dissemination Oriented Protocols

The majority of these protocols have been tree-based routing protocols. MintRoute [71] was one of the initial routing protocols for L2Ns. Starting with the gateway, or base station node, beacon messages are broadcast announcing the cost to reach the gateway, both in terms of hops, as well as ETX, or expected transmissions. ETX is calculated using the inverse of packet reception ratio (PRR) over a link, which is roughly approximated using an exponentially weighted moving average of received signal strength indicators (RSSI) for each packet. In subsequent versions for a popular platform, link cost estimation is done by using a chip link quality indicator (LQI), which is provided for every received packet. While MintRoute was successful, its main shortcomings stemmed from the limited sophistication of its link estimation and loss response techniques, which reduced efficiency in difficult RF environments.

The successor to MintRoute is CTP [24, 29]. CTP utilizes a more accurate link estimator, in which control and data traffic is used to provide link estimation, although two different link estimators are maintained. Furthermore, CTP allows multiple destinations to serve as data sinks, in which case multiple trees are formed with each destination serving as a tree root. Finally, multiple potential next-hop parents are maintained, although only one is used at any given time. In one of their tests, CTP displayed 97% overall reliability in a difficult RF environment, as opposed to the 70% reliability obtained by MintRoute [24].

CentRoute [65] provides centralized tree routing, allowing for dissemination of tasklets and collection of information in the Tenet [30] architecture. A single node serves as the sink, and other nodes in the network send join request messages. If the node is a direct neighbor of the sink, the sink replies with an accept message, and the node joins the tree

by selecting the sink as its next hop. Otherwise, if a node that is part of the tree overhears a join message, it forwards this message to the sink. When the sink receives a join message (originating from a non-neighbor node), it waits for the same join message to arrive along multiple paths and selects the best path. The sink then source routes the accept message to the originating node along this path, and the node appropriately sets its next hop. Once created, the tree is frozen, unless some threshold of failures occur, after which a node disengages and begins the join process anew. This approach is limited to single destination routing, and also affords no flexibility for dynamic topology conditions (unless links completely break).

2.4.2 Other Routing Protocols

BVR [26] is a geographical location-based routing protocol for L2Ns. A set of landmark beacons are selected whose identity is known by all nodes in the network. Every node is given a virtual coordinate, which is the vector of distances to all of the beacons in the network. When a node wants to send a packet to a destination node, it greedily forwards the packet along routes that move it closer to the destination. If it is unable to do this (i.e. gets stuck at some point), it forwards the packet to the beacon closest to the destination. In the case that this beacon is not able to use greedy routing to forward to the destination, scoped flooding is used to find the destination. As demonstrated in other papers [52], BVR suffers from poor transmission and routing stretch, particularly as the network grows in size. Variations to improve performance trade off resources by maintaining the entire 2-hop neighbor table in memory. Another difficulty is that in such a network, the identity of the beacons must be known a priori, and can not be modified dynamically. A BVR location service [56] was designed, but struggled because it still necessitated significant network overhead and state.

S4 [52], is designed to provide small stretch and state in order to enable scalable routing. It utilizes a modified compact routing algorithm tailored for L2Ns. Some set of nodes in

the network are designated as beacons (with $\sqrt{N}$ being the optimal number). A node n's local neighborhood cluster is defined as the set of nodes whose distance to n is less than or equal to the distance to their closest beacon. Each node maintains routes to all other nodes in their cluster, and all beacons in the network. A node can route to nodes outside its cluster by routing to the destination node's closest beacon. However, this feature requires a directory service that the node can contact to obtain this information, and furthermore a beacon selection mechanism is needed, both for initial beacon selection, as well as under failure conditions, and the number of beacons can not be easily modified during runtime. S4 has a theoretical worst case routing stretch of 3 (although published results indicate an average routing stretch of 1.2), and $O(\sqrt{N})$ state, although this does not account for local cluster management. In section 4.4, we compare HYDRO to S4, demonstrating that we achieve greater reliability, with smaller stretch, state, and control traffic.

One of the shifts in ROLL has been toward an IPv6 based architecture, centered around 6LoWPAN [55], the low-powered adaptation designed for L2Ns. Hui et al. [35] presented a complete IPv6 network architecture for L2Ns, which include a low power link layer, an IP-based network and transport layer, as well as high-level applications. Of particular interest to our work was Hui's contribution to reliable multi-parent tree-based routing, and confidence driven link estimation. This work served as the foundation and inspiration for many of our algorithms and has been invaluable in helping us to frame our architecture.

TSMP [60], which is incorporated in the WirelessHART standard, is a well-established industry approach that includes slotted time & channel scheduling using the global information, but little technical data is available for it. Furthermore, it operates at the link-layer, using channel hopping, and does not provide any routing capabilities (at least none that is specified in publicly available documentation).

2.5 Internet Routing Protocols

We discuss two separate bodies of work in this section: Centralized Internet Routing Architectures / Protocols, and IP-Based Ad-Hoc Networking Protocols(i.e. MANET [4]).

2.5.1 Centralized Routing Architectures / Protocols

Feamster et al. make the case for separating the control aspect of routing from routers by introducing RCP, the Routing Control Platform [23]. Designed for both intra-AS (autonomous system) and inter-AS routing, the motivation is that by distributing all aspects of routing, interactions became increasingly complex, heavy bandwidth usage results from topology changes, and forwarding loops and routing failures that are incredibly difficult to diagnose continually present themselves. To alleviate such difficulties, RCP serves as the primary interaction point for all routers within an AS. By participating in the network, RCP is able to obtain a complete view of the topology. A network operator can specify the routing policy at a single place (which typically is to simply mimic that of a fully-meshed iBGP AS), and then the RCP node advertises the correct route to each router in the network. RCP then serves the role of an egress router in communicating routing information with other AS's using eBGP (allowing for a smooth incremental deployment). As an ideal end-state, all AS's would have an RCP, and these RCP's would communicate with each other to exchange routing information.

Caesar et al. present an actual design and implementation of an RCP [15]. Leveraging the existing OSPF and BGP protocols in place in most existing networks, the implementation has three distinct components: an IGP Viewer, a BGP Engine, and a Route Control Server (RCS). The IGP Viewer obtains IGP topology information by building adjacencies with routers in the network. The BGP Engine learns BGP routes from the routers, and also communicates routing assignments to each router. Finally, the RCS uses the information obtained by the other two components to compute routing assignments according to some

specified policy, which currently defaults to full-mesh iBGP configurations. As a key point, to avoid creating a single point of failure, RCS nodes are distributed throughout an AS, but they are not precise replicas, and there are only loose consistency requirements, which are detailed in the paper.

While the creation of a distinct control hierarchy certainly permeates our design, the practicality of RCP for ROLL network is limited. Focusing on only the intra-AS aspect, it leverages full underlying IP and TCP connectivity, as well as OSPF and BGP for information dissemination, which neither exist nor are practical in ROLL networks. A key part of our design involves designing mechanisms that provide this underlying connectivity and accurately constructing and maintaining global topology views. Furthermore, full consistent connectivity is maintained by all nodes in the network at all times (often through flooding of Link State Advertisements), resulting in significant memory and network overhead, as opposed to on-demand schemes that are often a better fit for ROLL traffic patterns. The lossy and dynamic nature of low-power wireless would only exacerbate network congestion in such an architecture. Finally, their attempts at reducing overhead relies on prefix-based IP addressing, conflating node address and geographical location, while ROLL networks primarily embrace flat-label based addressing schemes.

Greenberg et al. present 4D [32], a general framework for separating routing from routers. The framework consists of four components: a decision plane, a dissemination plane, a discovery plane, and a data plane. The decision plane, composed of a single logical decision element (which is replicated for fault tolerance), serves as the *network brain*, serving as the interface for network-wide policy specification, which is subsequently communicated to individual routers and switches. The dissemination plane is tasked with creating the logical channels between the decision element and each router and switch, providing a medium for control information flow, independent of the state of the data plane. The discovery plane refers to the process in which each router/switch discovers its own capabilities, its neighboring routers/switches (including those in other ASes), and commu-

nicates this information across the dissemination plane to the decision element, allowing for the creation of a global view of the network. Finally, the data plane is greatly simplified, responsible only for forwarding packets according to the forwarding table entries installed by the decision element.

As a framework paper, 4D does not provide any specific instantiations of any of the components. Although seemingly targeted towards the general Internet, the same modularization applies in the context of ROLL as well; our routing architecture for ROLL, presented in chapter 3, can roughly be decomposed into the same four components. This componentization is particularly effective in highlighting the characteristics that differentiate ROLL from the general Internet. The dissemination plane has to be built over a series of lossy unreliable links, in which the *optimal* paths are potentially shifting frequently. Low power wireless also complicates the discovery plane, as there is a strong sense of dynamicity, even if the nodes are physically stationary, and also power duty cycles that reach 0.1% and below. Furthermore, energy constraints prohibit excessive control traffic (relative to very low data rates), which complicates communication of this discovery information to the decision plane. Finally, the data plane must provide certain reliability and delivery guarantees in the face of lossy and moving paths.

Yan et al. present Tesseract [72], a faithful instantiation of the 4D framework and concept. Switches use periodic beaconing to determine their neighboring nodes, as well as their own capabilities, and then transmit this information to a *decision element*, which forms the basis of the decision plane. Path explorer messages are flooded through the network to enable the building of the dissemination plane. Subsequently, the decision element sends all control messages to switches using source routes, and in turn, these switches reverse these source routes for sending information to the decision element. The decision plane is implemented as an abstract model, in which specific algorithms are separated from the underlying link technologies, to provide an element of portability. To provide robustness at the decision plane level, multiple decision elements are maintained at any given time, with

all information being fully replicated across all of them. However, only one of them is a master (i.e. active) at any given time, while the rest are only used in the case of a master failure. The master is chosen through a leader-election process and arbitrarily assigned priorities. Security also plays a large role in Tesseract, with numerous mechanisms to enable encryption, certification, and authentication.

Tesseract suffers from similar shortcomings (in the context of ROLL) to 4D, which we discussed above. In addition, they maintain multiple decision elements, but only keep one active, as the main goal is robustness. A single active point of control is impractical for L2Ns for scalability reasons, as deep networks, lossy links, and localized traffic patterns all suffer with this model. Furthermore, Tesseract strives to provide fully connected shortest path (or most efficient) based routing, which necessitates large amounts of control traffic (current implementation floods network every 20ms), and state at the nodes that scales with the size of the network. The dissemination plane relies on source routing in both directions, stripping the switches of any local flexibility to deal with unreliable links, and dynamic topology shifts, as is common in L2N networks. This results in additional control traffic for every instance of packet delivery failure, as does the need for full information at decision elements.

One paradigm for providing effective routing is to utilize meaningful, or routable, addressing schemes. One of the reasons that IP prefix-based routing is effective is that IP-addresses tie together a name and geographical location. In other words, except in very rare cases (such as Mobile IP solutions), all IP addresses with a shared prefix (of varying length depending on the size of subnet) are geographically co-located, allowing remote routers to maintain a single routing entry for that block of addresses. This reduces both memory state at routers, and the size of control traffic.

This conflation of address and location is difficult in L2Ns. We discussed examples of this in section 2.4.2 in the form of geographic routing and compact routing schemes. The challenge in implementing such a concept is the dynamic nature of the network. Varying

link qualities cause optimal routing paths to change, which would imply that the router's address would have to adjust accordingly. This would require a lookup service for nodes to report updates to, and request lookups from. This is in fact one of the main shortcomings of proposed geographic and compact routing schemes in the L2N space. Despite the shortcomings of existing solutions, this may be a potential avenue that warrants additional research.

Ethane [17], the successor to SANE [16] and the predecessor to NOX [33], presents a centralized architecture for providing high-level security policies. Designed for large enterprise networks, the network is divided into four tiers: a controller (or multiple ones for fault tolerance), switches, end hosts or servers, and users. Whenever a switch is connected to the network, it registers with the controller, and establishes a secure backchannel to it. When an end host joins the network, the switch also reports this registration, reporting the class of device and attachment port. Finally, users are able to associate with one or more end host machines. Consequently, the controller maintains a complete and accurate view of the topology. Network administrators specify high-level access policies, such as restricting which servers a certain user may communicate with. Each switch maintains a flow table against which it can classify incoming packets using the packet header's 10-tuple. If no applicable entry exists, the packet is forwarded to the controller. The controller consults its policy specification, and installs hop-by-hop flow entries along the path. HYDRO utilizes this concept of installing flow entries and forwarding packets belonging to unclassifiable flows to the controller.

Ethane assumes the availability of a secure and reliable channel to the controller, and also lends itself to trivial construction of a global topology view. Network switches all have ample bandwidth and memory, allowing Ethane to support 70,000 flows / second at line speeds. Finally, Ethane operates on top of layer 2 switching, which performs automatic failure rerouting to direct a packet to the addressed next hop. While we discuss Ethane's successor, NOX [33], further in chapter 6, we note that architecturally it is the same as

Ethane and hence suffers from the same limitations in the context of ROLL.

Multiprotocol Label Switching [62], or MPLS, allows for the creation of explicit paths in the network, often either to provide QoS or VPN services. Network operators install label entries at Label Switch Routers in the network, which dictate the action to take upon receiving packets with the given label. Packets carry a label, or even potentially a stack of labels, which are attached at Label Edge Routers, allowing for fast lookups at each Label Switch Router. MPLS on its own is simply a mechanism for installing, distributing, and forwarding based on labels; protocols for obtaining network topology information and dictating which paths to install are external to MPLS.

2.5.2 Mobile Ad-Hoc Networking (MANET) Protocols

There is also extensive literature on routing protocols for ad-hoc wireless networks and L2Ns in particular. The MANET [4] working group focuses on such protocols and issues.

AODV [58] uses flooded RREQ messages to discover paths to destinations on demand, with intermediate nodes creating hop-by-hop entries for bidirectional flows. DSR [39] is also an on-demand routing solution, but uses source routing to route packets. OLSR [19] is a link state algorithm that uses multipoint relays to reduce the flood of link-state advertisements.

These and other MANET protocols provide point-to-point routing capabilities in mobile ad-hoc wireless networks. Both on-demand and link-state based solutions exist, but the focus is on providing reliable any-to-any delivery in networks with mobile nodes that are resource rich. Consequently, most of these protocols have large control traffic and/or state requirements. The ROLL protocol survey [48] evaluated the RFC specification (or mature Internet-Draft) of the most widely used MANET protocols against the five ROLL criteria, demonstrating that in their current form no MANET protocol meets more than three of the five criteria. L2N specific implementations of some of these protocols exists, such as TinyAODV and TYMO, but have received limited exposure and acceptance due to their

poor (or non-existent) evaluations and limited feature-sets.

2.6 Summary

In this section we discussed the characteristics of Lossy and Low Power Networks (L2Ns), highlighting their low-power and resource-starved nature, as well as their typical application domains and communication paradigms which differentiate them from broader networking. We highlighted three key sets of items that any ROLL protocol will need to address:

- **Core Challenges (§ 2.2.1):** A set of 3 metrics that must be considered in the context of L2Ns.
- **ROLL Requirements (§ 2.2.2):** 5 quantifiable criteria requirements for a ROLL routing protocol.
- **Centralized ROLL Protocols (§ 2.3)** 3 factors that prevent Internet-based centralized routing protocols from being readily applicable to ROLL.

With these three sets of issues, we then explored existing protocols, both in the context of L2Ns, as well as centralized and ad-hoc routing protocols for the Internet at large, highlighting aspects that we built upon in our work and also reinforcing the need for our work and the differentiation with past systems.

Chapter 3

HYDRO: A Two-Tiered Hybrid Routing Protocol for L2Ns

3.1 Design Overview

In order to perform centralized routing, two logically distinct types of devices are utilized: nodes and controllers. Centralized routing occurs when the controller maintains the link state of the network and opportunistically optimizes paths by inserting routing table entries into routers. If no path has been inserted, traffic flows to the controller where it is routed appropriately. Nodes have small routing and forwarding tables, and obtain local connectivity information to participate in the routing protocol. Controllers also participate in the routing protocol, but have more local resources and often possess backhaul connectivity.

Many previous routing solutions for ad-hoc networks operate in a completely distributed manner. A key advantage of this design is to allow local flexibility to deal with link and node failures. A goal when designing a centralized routing protocol for wireless networks is to locate the sweet spot between *local flexibility* and *centralized control* so as to gain the small node state centralized routing provides, while remaining agile and allowing for local repair.

Providing centralized routing in an L2N requires three primitives. First, a stable "back channel" is necessary to allow nodes to communicate reliably with a controller. In L2Ns, this back channel is constructed from the same links as the operational network. Second, there must be a mechanism for discovering and collecting the local connectivity at each node and reporting it to form the global topology. Finally, the controller must have the capability to insert routing state into the network of node. Once these are in place, other decisions such as the forwarding discipline which determines how a packet arriving at a network node is dispatched, and a controller policy which decides on which routes to install may be examined.

There are multiple design decisions to be made about each aspect. In the following sections, we explore several possible design points and describe our motivation for choosing a particular set of solutions to evaluate. Our contribution here lies in the fact that we tackle these problems in a difficult environment. For example, discovering topology and connecting to a central controller are nearly trivial in a wired network where topologies are determined by physical connection and are unlikely to change quickly. Furthermore, the available bandwidth and memory in those setting is practically infinite in comparison to an L2N, and there is less incentive to reduce message complexity; in contrast, every packet sent in an L2N has a power cost which quickly becomes significant. We do not find it obvious that a centralized algorithm will be able to adapt to the low bandwidth and high churn of wireless networks.

In a nutshell, our solution uses a distributed algorithm to build a DAG rooted at the central controller, providing enough state for this controller to create a sufficient global view of the topology. As a baseline, this DAG provides centralized point-to-point routing by allowing nodes to route messages to the controller, which then source-routes them to the correct destination. Our protocol improves upon this baseline by having the controller install in-network routing entries containing optimized routes for frequently-communicating endpoints.

3.1.1 Default Route Establishment

Decision: Provide local flexibility by maintaining link estimates for multiple next hops toward a controller.

A stable default route to a controller must form the basis of any centralized routing scheme. Without such a backbone, it is impossible to communicate topology information to and receive routing information from the controller. When designing a centralized protocol, this is the level at which to incorporate the goal of local flexibility. By doing so, we are able to maximize the reliability of this channel to the controller, which will result in improved performance for the rest of the protocol.

Building a reliable path to a controller is also advantageous given the expected traffic patterns of many L2Ns. Often, these networks function as access networks, with an edge router providing routing to other networks. In the common case where this edge router is also the controller, we ensure reliable access to the external network.

This task is a place where we can consult the literature; much work been devoted toward collection protocols where all data is forwarded toward a set of sinks. Common techniques we use include estimating the quality of links using the expected transmissions metric (ETX), and proactively maintaining a path through the use of beacons and an additive gradient. Furthermore, introducing a small set of candidate next hop nodes in the direction of a collection point has been shown to improve reliability, especially if the protocol periodically switches between them so as to maintain up-to-date link estimates.

In our solution, we take advantage of the distributed algorithm developed by Hui et al. [34] to maintain the set of candidate neighbors in the direction of an controller. We also maintain both short- and long-term estimates of each link. The long term estimate is used as a stable metric on which to form routes, while the short term estimate is used to provide quick detection of link or node failure. We also maintain a measure of our confidence in the link estimate. In our current implementation, this is simply the number of times we have tried to use the link for unicast traffic. We refer the reader to the body of work on

understanding low-power wireless links for additional detail [64, 25, 63, 35].

Due to the low data rates common to L2Ns, we seek to eliminate extraneous beacon messages; conventional routing beacon rates would dominate the traffic rate. As a result, we do not rely on periodic beacons for path formation and neighbor discovery. In our design, IPv6 Router Solicitations and Router Advertisement messages are used instead. Router Advertisements are only sent

- In response to a solicitation
- If the path metric or hop count changes beyond a threshold.

Solicitations are only sent when no default route is available. The advertisement messages carry the hop count and metric to a controller from the advertising node. Each node uses the information in these messages to build its neighbor table and choose a route toward the controller

One critical detail is allowing reverse communication from the controller to the network nodes. Possible solutions include storing reverse path information at intermediate nodes, or source routing packets from the controller. We reject storing reverse paths pragmatically due to its unbounded state, and philosophically because in a centralized scheme, it would represent an unnecessary dispersion of state. Thus, we choose to source route all packets from the controller into the network.

3.1.2 Global Topology View

Decision: Report ETX link estimates of only frequently-used links.

Decision: Piggyback most topology reports on data, but allow triggered updates for reactivity.

The overall goal of topology collection is to give the controller a global view of the network. This has clear similarities with link-state algorithms, with the critical difference

being that in a centralized algorithm the topology is reported only to the controller. The two facets of this problem are deciding what to report, and when to report it.

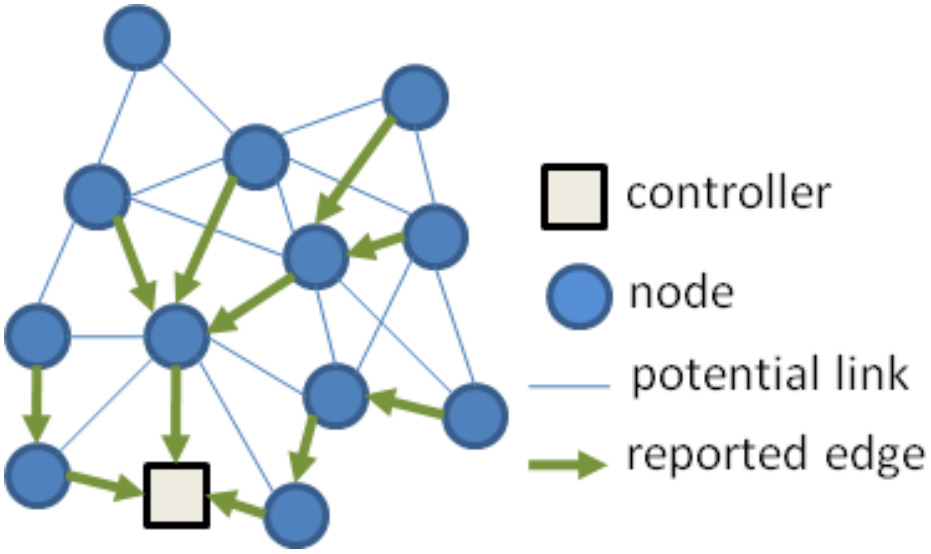


Figure 3.1: An example collected topology when only one link is reported by each node. By default we report four links per node.

A basic question is *which* links to report. A node must evaluate neighbors within communication range and select a subset of them to report to the controller as available for routing. One strategy would be to have each node report its full neighbor set to the controller, so as to provide a complete picture of the topology. However, this is infeasible because even with a fixed-size neighbor table, the amount of state required to be reported is still likely to be overly large. More importantly, the node is unlikely to be able to maintain reliable link estimates to more then a few different nodes at a time. Due to these limitations, our design decision was to report the links used to communicate with the controller, and devise a policy to ensure that more then one link is used as a default route so as to maintain good estimates of a few links. In addition to reporting the existence of a link, a node should also report a link metric to allow the controller to choose good links. Figure 3.1 depicts a scenario in which each node only reports one link.

This decision to report links in the direction of the controller impacts the optimality of the routes the controller will be able to form. We recognize that this strategy may lead to

sub-optimal routes, and discuss it further in the evaluation.

The second question is *when* to report. The dual goals here are to maximize responsiveness by ensuring the controller is notified of changes in a timely fashion, and to minimize control traffic. Link failures are more important to report then new links, since as long as the controller believes a failed link is present, it may attempt to use it which will lead to expensive rerouting. The appropriate techniques to apply here are triggered updates and piggybacking. When a link failure is detected by the link estimator, it immediately enqueues a notification to be sent to the controller as soon as possible. In order to minimize the overhead of the protocol, updates should be added to existing data traffic wherever possible. We insert a topology report into a data packet whenever a packet is sent along the default route and doing so would not cause segmentation at the adaptation layer.

3.1.3 Centralized Route Install

Decision: Primitive functionality allows inserting flow table entries into nodes' tables.

Decision: Installed state is either full routes at the flow source, or entries at each hop.

The third necessary component of a centralized algorithm is the ability to install routing state into the network. In this section, we discuss the options for doing so.

Each node must have a routing table which provides instructions on how to route each packet originated or being forwarded. Each routing entry has two components: the *flow match* and the *routing action*. The flow match provides a mechanism to classify outgoing packets by looking at elements in the packet header. In our implementation, the entire chain of headers is available to the routing engine in order to make a routing decision, although currently only the destination field of the IP header is used.

Two basic types of routing actions must be considered: either storing the next hop for a packet matching the flow match entry, or storing the entire source route. We call these choices *hop-by-hop*, and *source*, respectively, and implement both.

With hop-by-hop, the routing entry simply lists the layer 2 address of the next hop along

the path to the destination. The advantage of this approach is that it allows the sharing of entries between different overlapping paths. One disadvantage is that the state for a single path is distributed across the entire path, occupying routing table entries at all intermediate nodes, in addition to the end points. Through either routing table overflow, or node reboots, this can lead to state inconsistencies in the network and potential loops. Regardless, most existing centralized solutions utilize this approach.

With source, the originator of the packet places a source header in the packet and intermediate nodes simply forward it to the next hop according to the source header. This approach eliminates the possibility of inconsistent state along a path as it localizes all state at the source. The disadvantage of source-routing packets is that each routing entry is larger because it stores the full path and there is a per-packet penalty of carrying the route. Furthermore, repair in response to loss of a link may have to occur at many sources.

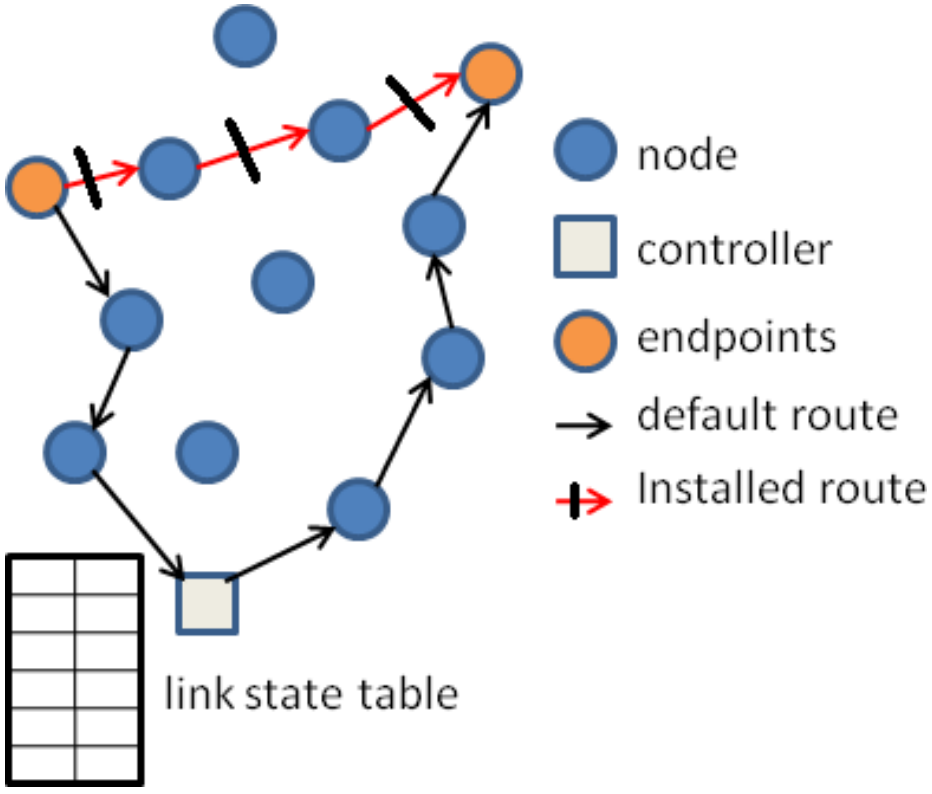


Figure 3.2: The effect of the route install primitive.

It is also necessary to provide a method for the controller to install and update the entries

Table 3.1: Key design questions and our decisions

Question	Decision
How are local flexibility and centralized control balanced?	Nodes maintain redundant paths to a controller. Packets are sent along this route if installed state is faulty.
What links does each node report?	Only links which the node has a high confidence estimate of.
How does the network respond to link and node failures?	A reactive, short-term link estimate triggers a message to the controller about a broken link after the link is used.
How are flow table entries inserted?	The controller inserts either full source routes or flow entries at each hop by adding the command install to data traffic.
How are forwarding decisions made?	Installed state is used if available. If unavailable, or fails, the default route is used.

in a node's table. We provide an IP extension header which contains a single hop-by-hop or source install message that can be piggybacked on a data packet to the node.

In the case when a message is sent from one node to another via the controller, the header installing the reverse route can be added to the message as it transits the controller thus avoiding an additional message. If that node then replies to the original node, it can then send the reply along the newly installed route, and install the opposite direction as it goes. This method has the benefit of not incurring any additional control message transmissions, although it requires some per-packet processing. Alternatively, the controller may generate a new packet back to the source containing a route to the destination.

Although the assumption of bi-directional traffic may not hold in all cases, it is quite common for point to point. Any TCP traffic will necessitate bidirectionality, and application level replies are likely for UDP traffic, as in RPC. The bidirectional nature of our link cost estimates imply that the best path from B $\rightarrow$ A is the exact reverse of A $\rightarrow$ B, allowing us to specify concurrent installation of the reverse path. The controller has to flexibility to select a particular (or multiple) route-installation model.

3.2 Our Hybrid Approach

Decision: The default route is used as a backup in the case where installed state is invalid.

Having built up the three primitives of a back channel, topology collection, and route install, we must now actually build a routing protocol. First we briefly consider the actions a node takes to make a forwarding decision. While there is not a great deal of design space here given the framework thus developed, it is worth noting.

Whenever a node is sending or forwarding a packet, a node gives priority to any information in a source header. If no source header is present, the node attempts to classify the packet against its routing table, and if no match is found, the packet is sent along the default route to the controller. If a next hop is present in the table, the node will first try the installed next hop and subsequently re-route the packet along the stable back channel if the installed route fails, in the hope that the controller will have another path to the destination. Furthermore, the failure will likely cause the eviction of the link in question from the neighbor table and a differential topology report due to the drop in the short-term link metric.

3.2.1 Controller Design

Definition: The controller's *install policy* determines when routes are installed in the network.

Definition: The *path policy* finds paths based on link qualities and node attributes.

It is only after we consider all the preceding issues that we can consider what centralized routing typically involves: how the controller should be designed. The high level question here are "when to install a route," and "what information to maintain."

In theory, the controller can maintain arbitrary information about the network. It clearly must maintain the link state to be able to generate routes, but beyond that there is a wealth of information which could be useful. Our philosophy is that to the extent possible, the con-

troller should be stateless, because it facilitates failure recovery. Where state is maintained, the state should be soft; the link state is an example of soft state since after a crash, it only needs to listen to topology reports for some period to recover the network topology. Other information we can envision being useful for routing decisions are statistics on active flows in the network and node characteristics for constraint-based routing. We leave hard-state as a last resort.

The *install policy* determines when a route should be installed in the network. A great deal of potential is available here for workload-specific optimization so as to avoid unnecessary route install messages. Simple policies could include only installing routes for TCP flows which are guaranteed to be bidirectional, or using an application-layer packet classifier to determine whether a packet is part of a long-lived flow. The controller should attempt to avoid sending install information for flows which consist of only a single packet. For our preliminary evaluation, we implement a simple policy where the first packet of a flow generates a message installing a route between the endpoints. Certainly, many more sophisticated techniques could be considered to improve the solution, if the overall approach is sound and the simple policy experiences significant overhead.

When the controller makes the decision to install a route, it must calculate the best path between the two end-points. Defining *best* is another component of an install policy. Our algorithm currently uses the simplest approach: it finds a minimum-ETX path from the source to destination using Dijkstra's algorithm. More complex algorithms, such as energy-based routing, history-based routing, and other policy-based routing algorithms can be specified, and we explore these in more detail in section 3.3.

If the controller determines that it is an intermediate node along this *best* path, it simply forwards the packet by source-routing it to the destination. This is also the case for packets from an external network which are directed toward some node in the subnet. We note that all messages from the controller are source-routed.

3.2.2 Shortcomings of Centralized Protocols

Introduction of a centralized element of routing raises a family of concerns. We briefly examine the most common ones raised in the literature and anecdotally in the context of our setting and approach.

Latency of sending packet through controller: The latency incurred by sending a packet through the controller, rather than directly, is typically only tens or hundreds of milliseconds, and often is only incurred on the first packet of a flow. More importantly, in the majority of L2N deployments, latency, particularly on the order of milliseconds, is not significant. There do exist real-time scenarios in which latency is critical; in such cases an on-demand solution like HYDRO may not be appropriate without quality-of-service adaptations.

Controller becomes a single point of failure: The failure of the controller would cripple the network, but this is likely true whether or not the controller performs significant routing functionality, as most L2N deployments use a controller as the egress point or sink for data collection. Also, we anticipate maintaining multiple controllers for fault tolerance and scalability as we seek to maintain a constant ratio between the ratio of nodes in the network and the number of controllers. We discuss this further in section 3.3, but for simplicity focus on single-controller networks until then.

A persistent back channel to the controller is needed: The directed acyclic graph rooted at the controller forms the underlying core of our protocol, and so if this were to break, our performance would suffer badly. However, collection-oriented routing is essentially building this back channel, and after nearly a decade of research solutions have become stable and resilient. In this vein, our algorithm provides swift local recovery to enable adaption to dynamic topological conditions, and also maintains multiple options for additional reli-

ability. Our results in section 4.1 demonstrate the reliability of this back channel.

Difficult to maintain consistent global view of topology: This constitutes one of the most difficult challenges, particularly in dynamic L2N networks where control traffic must be bound by the data rate to conserve energy. As discussed previously, we do not attempt to maintain a view of the entire topology, rather just a subset that is *good enough*. Fluctuating link qualities make our approach of only providing *confident* estimates crucial to reliability, and our mechanism for selecting bidirectional links allows us to cut the size of topology reports in half without losing information. In section 4.1, we provide initial evidence that it is possible to react sufficiently quickly to network changes.

3.2.3 ROLL Requirements

We briefly revisit the ROLL requirements to demonstrate that our design meets these requirements.

Table Scalability: State in our design is stored only for active flows. Whether it is installed at each hop or at the source, the total state resulting from that flow is the same.

Loss Response: The failure of nodes or links cause traffic to be diverted to the controller, which is simultaneously alerted of the change. Fresh state is installed once the controller rebuilds its topology. This traffic involves only the communicating nodes and the controller.

Control Cost: We have no periodic beacon traffic. Topology collection is data driven in most cases.

Link Cost: Paths are chosen based on ETX.

Node Cost: Constraint- and attribute- based routing are a topic of future work, but can be easily implemented as controller policies. Choosing the default route to respect node costs may require modifications to the distributed default route selection algorithm.

3.3 Extensions and Future Work

Section 3.1 outlined the design of HYDRO by focusing on the primary set of design decisions that formed the core routing protocol, the majority of which are realized in our initial implementation, and their performance examined in chapter 4. However, this section explores a set of concepts whose design is relatively mature and which will be incorporated in future releases of HYDRO.

3.3.1 Multiple Controllers

While our evaluations showed that HYDRO performs very well in a network of 225 nodes, certain problems begin to arise as the network gets bigger. Namely, as the network diameter grows, the cost of routing an initial packet to the controller increases, and the path between end points grows, leading to larger routing entries and increased overhead for source-routed packets.

Fortunately, as the network scales, the number of Tier 2 devices (Gateways/Controllers) typically scales accordingly, roughly maintaining the same ratio of Tier 1 devices (Nodes) to Tier 2 devices. Given that any Tier 2 device can act as a controller, larger networks will have more control points. Each controller advertises in the same fashion as the single-controller model. Nodes in the network select default routes that provide the lowest cost for reaching *A* controller; they are agnostic to which particular instance. In fact, nodes that have equidistant controller instances available could potentially have default routes that lead to different controllers.

The set of controllers are expected to form a connected graph through a secondary interface, typically the network interface that connects them to the network backbone. If the controllers connect to the backbone through a wired interface, this is trivial. However, in certain cases the controllers' connectivity will come through multi-hop mesh routing, in which any of many available protocols can be used to provide connectivity. In our research

we have modified a Meraki Mini wireless router to serve as a controller, instrumenting it with a second, low-power radio, as this form factor facilitates the practical deployment of controllers.

Our algorithm requires that each controller maintain a global view of the entire topology, i.e. the topology is replicated across all controllers. In other words, when a controller receives a topology report, it multicasts this to all other controllers. Consequently, when a node sends a packet through the default route, the controller is able to calculate the best path to any node in the network.

This setup has two implications:

- The cost of routing to the controller will remain stable despite the size of the network.
- Controllers can route packets directly through the destination's closest controller, rather than through the entire network.

3.3.2 Multicast

There are numerous methods for implementing multicast support, but we focus on a single one here. We assume that a node is aware of what multicast groups it belongs to, and has also reported this information to the controller. As background, Trickle [47] is a dissemination protocol that uses polite gossip and suppression to disseminate data through a broadcast-medium-based network quickly, yet efficiently.

When a node wants to send a multicast packet, it forwards it along the default route to the controller. The controller analyzes the topology to identify connected subgraphs within the network of members of the multicast group. It then chooses two members of each group (for robustness), and source routes the multicast message to them. When one of these members receives the multicast message, it starts a Trickle dissemination cycle to distribute it among other members in its connected subgraph.

As an extension to this work, the controller can potentially dictate for a node to join a specific multicast group, as it may serve as a bridge between two disjoint subgraphs.

3.3.3 Policy-Based Routing

Our algorithm currently selects the best route by running Dijkstra's algorithm over a graph with reported link costs. However, there are a host of other potential algorithms that can be used:

History-Based Algorithm: The controller can maintain statistics on which flows have been installed, including when installed routes have failed (this can be detected by looking at the invalid source-routing header after it has been sent through the default route). It can then decide to load balance flows to provide equal resource utilization, or route around links that seem to be getting congested. Also, as an optimization, the controller can maintain state on how many packets in a flow have traversed through it, holding off on installing a route until some threshold is reached that implies a long-lasting flow.

Energy-Based Routing: One of the features of HYDRO that we did not mention is the ability to include node attributes or properties along with the topology report. As one example, L2N devices are increasingly equipped with energy-meter technologies [37, 21], and so a node can report its remaining energy with each topology report. The controller can then take this energy into account when selecting a path, carefully avoiding nodes with depleted energy reserves.

3.4 Design Limitations

HYDRO is certainly not without its limitations, and while some are implementation-dependent, others are inherent to our design.

Highly Dynamic Networks: This is probably the most significant of the shortcomings. The local flexibility afforded in the default route establishment process is designed to accomodate the natural fluctions and vagaries of low-power wireless communication. However, for anything other than routing along the default route, the controller must first receive information about a topological change before it can install a route, which incurs some latency. In order to mitigate this, we use a short term estimator to detect drastic variance in link quality, and also expire state at the controller when necessary, which led to good performance in simulations where we failed an increasingly large number of nodes. However, it is not quite clear how fast the algorithm will adapt to node's moving and highly dynamic networks.

Single Point of Congestion/Failure: We argued earlier that all two-tiered hierarchies are susceptible to having the sink or egress device fail, and we actually deal with this problem through replication of controllers. However, these controllers also serve as a focal point for congestion. The mediocre performance of triangle routing was attributed to congestion and queue drops at nodes near the controllers as the load increased. However, hop-by-hop routing's performance went over a cliff as it had the same symptoms as triangle routing, except that when messages got through, they created additional control traffic in the form of route install messages, further congesting the network. Source routing performed well in these scenarios, but in cases where there is a sudden spike in interference, or a large set of temporary failures, similar patterns could emerge. Meanwhile, distributed solutions do not share this single point of congestion.

Difficulty with Longer Paths: As we mentioned in the motivation for multiple controllers, our algorithm suffers with longer paths because state, control costs, and even stretch (because of the initial triangle routing) increase. In order to mitigate these costs, we are investigating a hybrid approach in which source routing and hop-by-hop install are merged. The maximum length of a source-route entry will be capped by a threshold, T, and at every hop that is a multiple of T along the path, a source-route entry for the next T hops would be

inserted. For example, if T is set to 10, and the length of the path is 35 hops, the source will hold a source route for hops 1-10, hop 10 will have a source-route entry for hops 11-20, and so forth. This does not mitigate the longer triangle routing cost, but our multiple controllers solution addresses that.

3.5 Summary

In this chapter we presented the design of HYDRO, our hybrid routing protocol for L2Ns that brings together *centralized control* and *local flexibility*. For each component of our design, we stated the high-level design decision(s) behind our approach, and then proceeded to flush this out in greater detail. We highlighted the main shortcomings of centralized solutions and discussed how our solution overcomes these challenges, and also how it meets the five ROLL requirements that we had discussed in chapter 2. Finally, we highlighted some relatively-mature extensions to the work that are being rolled into the next revision of HYDRO, and closed with a discussion of HYDRO's limitations.

Chapter 4

HYDRO Evaluation

4.1 Evaluation Methodology

The goal of our evaluation is to provide evidence that the design as presented performs well in a range of situations that are likely to arise in practice. While a complete, systematic evaluation of this design's generality and performance is beyond the scope of this paper, we believe these initial results are promising enough to merit further work.

In order to evaluate the performance of HYDRO, we have tested it across a variety of target environments, with a range of options along multiple dimensions. For our initial, baseline testing, we implemented HYDRO in MATLAB, which simulated an ideal environment with no congestion, no collisions, and also utilized a unit-disk radio model.

We have also implemented a fully-functional implementation of HYDRO in TinyOS [46], the de facto operating system for L2Ns. As discussed previously, the core of our implementation is an IP-based tree-routing protocol, `b6lowpan`, which we have made publicly available under a BSD license. This includes code for the controller, which is currently a linux-class device with an 802.15.4 radio interface, and a network stack for the embedded devices.

This implementation enables two additional testing environments. TOSSIM [45] is a

packet-level discrete-event simulator for TinyOS applications that can create a range of topologies, and uses real-world noise traces to model low-power wireless communication. In addition, we utilized the Smote testbed at UC Berkeley's Soda Hall (figure 4.1), which is composed of 79 MicaZ motes (4KB RAM, 60KB ROM). At full power, the network is about 6 hops in diameter.

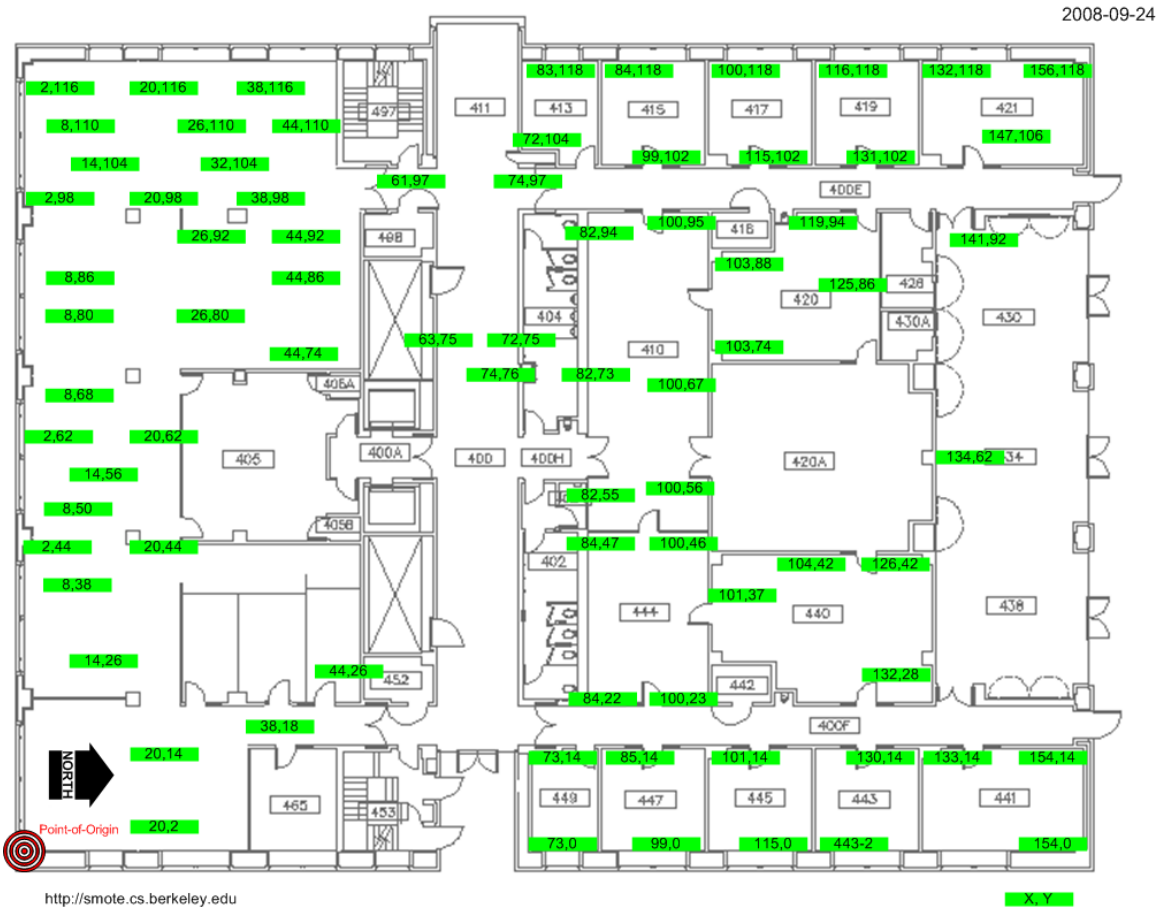


Figure 4.1: 79-node Smote Testbed at UC Berkeley

Given that there is no well established default traffic load or deployment configuration for L2N deployments, we have attempted to explore a variety of points across a number of dimensions. However, the bulk of traffic is expected to be collection and dissemination originating at or destined to an egress point in the network. We perform well on that workload without our centralized extension, and so we focus our evalution on the more difficult challenge of optimizing point to point. All testbed and simulation studies presented

in this section were conducted with concurrent, periodic data traffic from every node in the network to the network controller so as to simulate a working collection network with point-to-point routing added.

- **Traffic Paradigm:** The two traffic paradigms that we have tested are uniformly distributed source-destination pairs, and localized destinations, in which destination distances are limited to a certain number of hops.

- **Traffic Load:** We measure traffic load by the number of concurrent flows that are operating at any given time in the network. This tests the protocol's resilience to increased congestion, collisions, and also potential thrashing of routing-table entries. Most of our flows are actually bidirectional, simulating a regular TCP link. We use ICMP echo (ping) packets in our TOSSIM and testbed environments.

- **Network Size:** Our network is 79 nodes spread over office and labs with a diameter varying from three to six hops depending on the environment. In simulation we used 225 nodes in both the MATLAB and TOSSIM environments. Our claim of scaling to much larger networks stems from our belief that this scalability is relatively trivial when introducing additional controllers; all our experiments focus on a single controller. This means we can theoretically support 10000 nodes with 45 controllers, while S4 would recommend 100 beacons.

- **Topology, Density and Network Diameter:** The two topologies we have experimented with are a grid, and uniform distribution. Along with the physical size of the target environment, these also determine density and network diameter. High density challenges our ability to maintain a consistent global view and can lead to additional congestion/collisions, while large network diameters decrease the likelihood of end-to-end delivery success.

- **Link and Node Churn:** On the testbed, link churn is experienced in an actively used workspace with extensive wifi. In simulation, we shut off an increasing number of

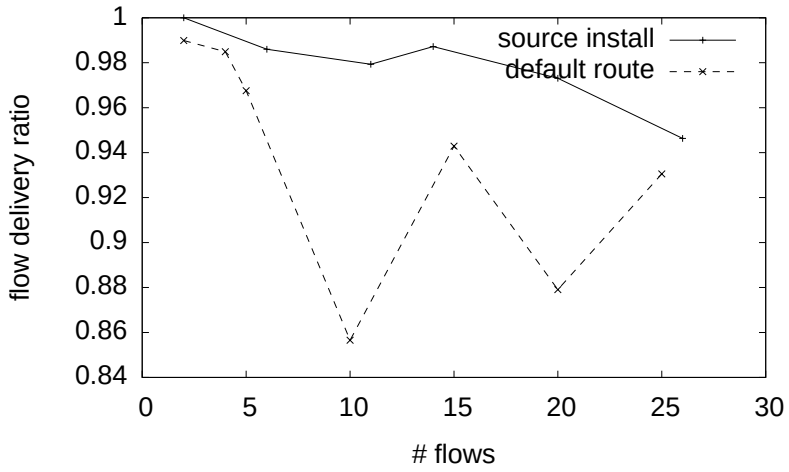
nodes in the experiment, forcing HYDRO to adapt. Link churn refers to the temporal fluctuation in the quality of a single link, which is a common phenomena when the SINR ratio is just above the threshold. We simulate this by lowering the radio power enough to cause bi-modal links.

In order to provide a thorough stress-testing, in many cases we attempt to push parameters values to an extreme in order to discover levels at which HYDRO's performance degrades significantly. However, we also point out where along the spectrum typical applications lie in order to give a better sense of HYDRO's anticipated real-world performance.

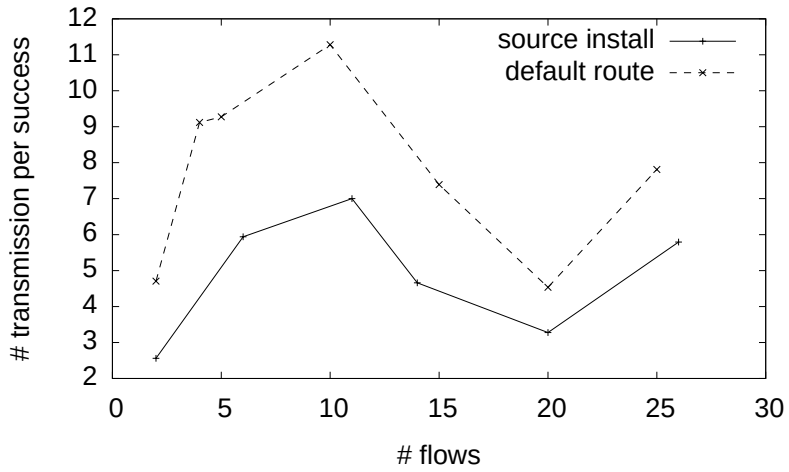
4.2 Testbed Evaluation

We begin our design evaluation with actual testbed results, shown in figure 4.2. For this experiment, we installed our implementation onto the testbed and started multiple flows each with a rate of one packet every two seconds, between random sources and destinations. Intermediate nodes logged information about each transmission to a serial backchannel, and install messages were accounted to the flow which caused them. For each different number of flows, we reconstructed the number of link-layer transmissions required to deliver a successful send, and the fraction of packet originations which were successful. We compared the effect of using installed paths with the cost of going through the controller for each packet; it is clear that involving the controller in every flow reduces reliability. It also increases stretch, as flows routed through the controller needed roughly twice as many transmissions per delivered packet than flows using installed state.

In order to further stress the protocol, we reduced the transmission power to -22dBm. Due to the topology of our testbed, only half of the nodes remained connected, forming a 4-hop network with 25 nods. The rationale for this experiment is to reduce the received power so as to increase the radio's sensitivity to interference and noise. The results from this experiment are shown in figure 4.3. It is clear that as the traffic load increases, more packets



(a) Packet Delivery Rate



(b) Transmissions per Packet

Figure 4.2: Smote Testbed (50 nodes) - bidirectional ping flows

are dropped. However, the network performs well with 5 flows involving 10 nodes, or half of the network. Furthermore, given the small size of the network, the mean transmission count appears reasonable.

4.3 MATLAB Simulations

Real world performance is dictated by numerous factors that cannot easily be controlled. To better understand the underlying interaction and to examine settings that are hard to establish empirically, we perform matlab simulations under a highly idealized model. All

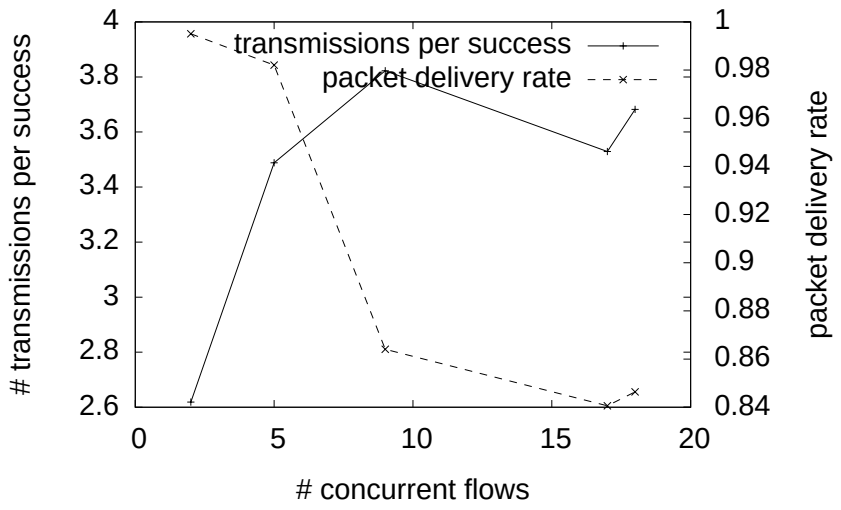
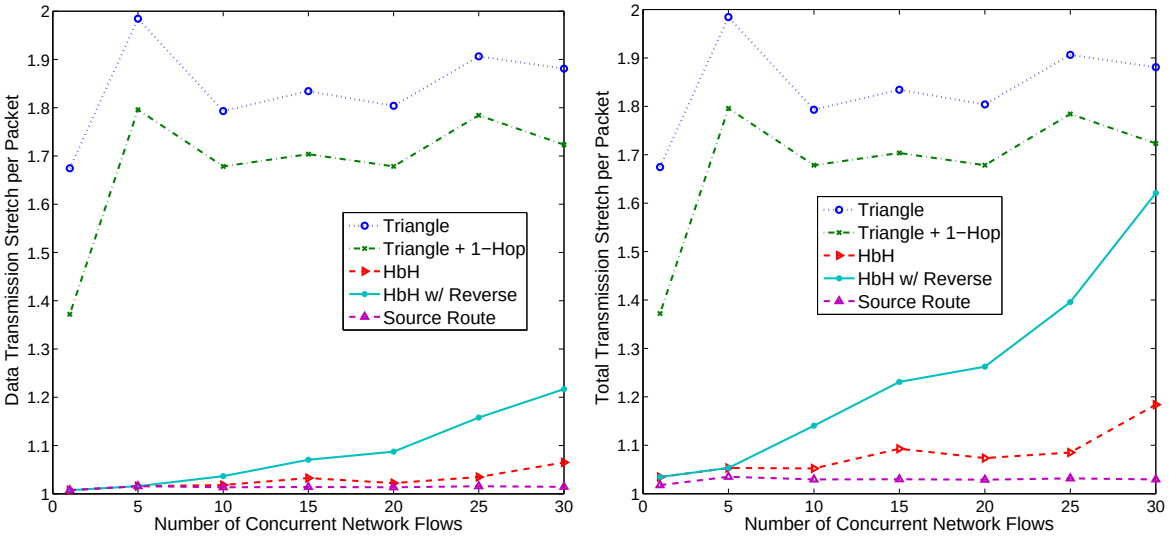


Figure 4.3: 25 nodes of the Smote testbed at -22dBm

nodes operate with a unit-disk radio model with a communication range of one unit. For simplicity, topology reports are not considered and the controller is assumed to have a complete view of the entire network. There are no packet losses in this network. Each simulation is run ten times and the results aggregated.

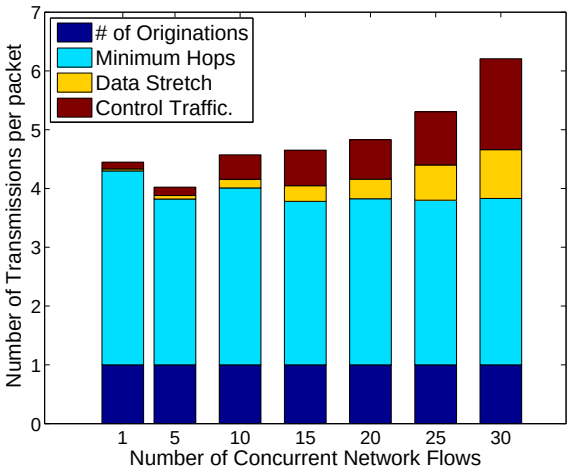
Figure 4.4 shows the results of our initial simulation in which 100 nodes (including the controller) are randomly placed in a 5 x 5 grid. The average node density was 10.4, and the size of the routing table is six entries, unless otherwise noted. We have plotted the results for 5 different algorithms:

- **Triangle:** All packets go through controller.
- **Triangle + 1-Hop:** Same as Triangle except that packets destined to one-hop neighbors are directly delivered.
- **HbH Install:** HYDRO with the Hop by Hop Install option.
- **HbH Install w/ Reverse:** Same as previous, except both directions of flow are installed.
- **Source Install:** HYDRO with Source Route Install option.



(a) Data Transmission Stretch

(b) Total Transmission Stretch



(c) Hop by Hop w/ Reverse Breakdown

Figure 4.4: 100 Node Uniformly Distributed Network

- **Optimal:** Optimal shortest-path algorithm.

The length of each flow in this simulation is 50 packets.

Figure 4.4(a) shows the transmission stretch[1] per packet when only data packets are considered, while figure 4.4(b) consider all packets, including route-install messages.

We see that triangle routing provides an upper bound on data transmission stretch at roughly 1.85, although when communication with direct neighbors is allowed, this bound

[1]Since there are no packet losses, or multiple path forwarding, the routing stretch and transmission stretch are equal in these simulations.

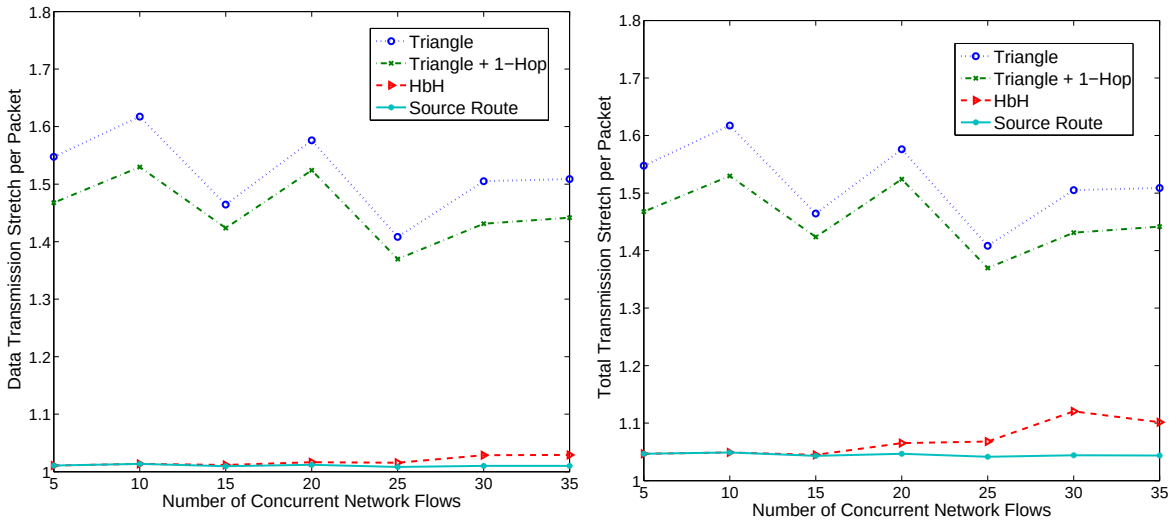
falls to about 1.7 with this particular traffic load. HYDRO with the source route install performs quite well, with a maximum data stretch of 1.02, and a maximum total stretch of 1.03, remaining stable despite an increase in the number of flows. However, we note that these stretch results are somewhat optimistic in the case of triangle routing, since stretch can become arbitrariliy large in the case of local communication far from a controller.

The Hop-by-Hop install options for HYDRO degrade in performance as the number of flows increases. The main reason is that when intermediate nodes must maintain routing state, their state requirements become $O(ND)$, where ND is all destinations accessed by the network, not just that particular node (and is bounded by the number of nodes in the network). Consequently, the routing table overflows and thrashing occur as destinations are uniformly distributed with this traffic load. Hop-by-Hop with reverse exacerbates this problem, because state now becomes $O(ND + S)$, since two entries are being installed at each intermediate node. Finally, the control cost of hop-by-hop is higher than source install because the route install message has to traverse the entire path, rather than be constrained to only the source.

Figure 4.4(c) provides a detailed breakdown of all the components of transmissions for Hop-by-Hop with reverse across increasing flows. As expected, we see that both data transmission stretch, and the control costs are increasing rapidly. While data transmission stretch is bounded by Triangle + 1-Hop, in the worst case a route install message could be sent for every packet, creating a much larger total transmission stretch than any of the other algorithms.

Figure 4.5 demonstrates a slightly different setup, in which 225 nodes are arranged in a 15 x 15 grid, with the controller placed in the center. We cut the length of the flow in half, although again we feel this is a conservative estimate. The average density of each node in this topology is 4. Also, all flows are bidirectional, simulating a TCP link. Subsequently, both the Hop-by-Hop and Source-install options install the reverse route concurrently.

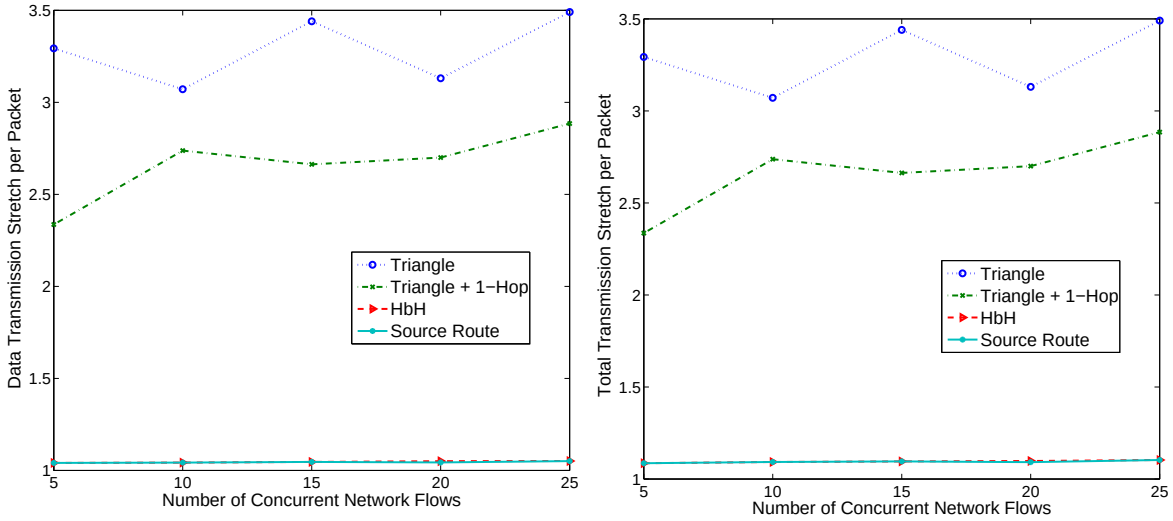
The larger network provides longer paths between source-destination pairs on average,



(a) Data Transmission Stretch (b) Total Transmission Stretch

Figure 4.5: 225 Nodes Arranged in a 15 x 15 Grid

explaining the drop in the triangle routing variations' transmission stretch. Source-install remains stable with a data stretch of 1.01, and total stretch of 1.05, as the benefits of the longer paths are offset by less packets to amortize the control cost over. Hop-by-Hop demonstrates the same thrashing behavior as with the previous workload, indicating that despite changes in density and network diameter, there will still be the same overlap and poor behavior.



(a) Data Transmission Stretch (b) Total Transmission Stretch

Figure 4.6: 225 Nodes Arranged in a 15 x 15 Grid with Local Destinations

We reran the same experiment, except using local destinations, where destinations could not be more than 5 hops away, and the results are shown in figure 4.6. This is a common workload, as for example in a building automation deployment, the network spreads out throughout the entire building, yet it is much more common for nodes on a given floor, or geographically similar area to communicate regularly.

There are only two main differences: the stretch of the triangle-oriented algorithms are much greater, because the length of the path to the controller has not changed, while the path between a source and destination is now much shorter on average. Second, hop-by-hop performs the same as source-install. With the shorter path lengths and distributed workload, there is a much smaller chance that two paths will utilize the same intermediate nodes, and so avoid the thrashing issue, or rather delay it until there are a larger number of concurrent flows.

While the simplified operating environment limits the applicability of these results to the real-world, they do show the fundamental contraints at work. While HYDRO with a source-route install option seems very promising, the hop-by-hop install option appears susceptible to variance in traffic degree, and a poor choice for these target environments.

4.4 TOSSIM Evaluation

TOSSIM allows developers to create topologies for running TinyOS applications in a discrete event simulator, which models network traffic at a packet-level, rather than a bit-level. In order to provide the most realistic simulation of wireless radio characteristics, measured noise traces are used to develop a noise model to more accurately simulate radio behavior.

Although a user can record custom noise traces from a desired environment and feed that into the simulator, TOSSIM itself provides two sample noise traces that can be used. One is a *good* trace in that the noise floor is low and has little variation. This trace does not typically result in lossless links; rather we often see a burstiness of losses. The other is a

poor trace, with a higher noise floor, and more temporal variation in the noise level.

The topology for our TOSSIM experiments is the same 15 x 15 grid from the MATLAB experiments, with the controller placed at the center. The density at each node varies, as the communication radius can range from 1 to 2 units, and the unit-disk model is no longer being used. Our traffic type is bidirectional IPv6 Ping packets, and the sending rate is 1 packet every two seconds. Unless otherwise noted, the length of each flow is 50 packets, for the same application-driven reasoning described in section 4.3. The size of each node's routing-table, where appropriate, is set to accomodate at least $O(D)$ entries.

The first study we undertake in figure 4.7 shows the performance of our underlying default route in both the good and poor environments. With nodes reporting data every 45 seconds, we achieve a delivery rate of greater than 99% from all nodes over several hours. We compare our results to those of CTP, a widely- used collection protocol available in TinyOS [24], and with our protocol on our testbed. Our underlying DAG is considerably more reliable then CTP in the controlled simulator, and in the real world testbed it is more reliable still. Our goal is only to demonstrate that a sufficiently reliable back-channel is present.

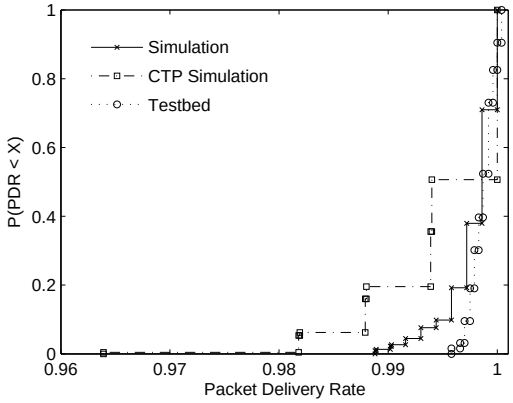
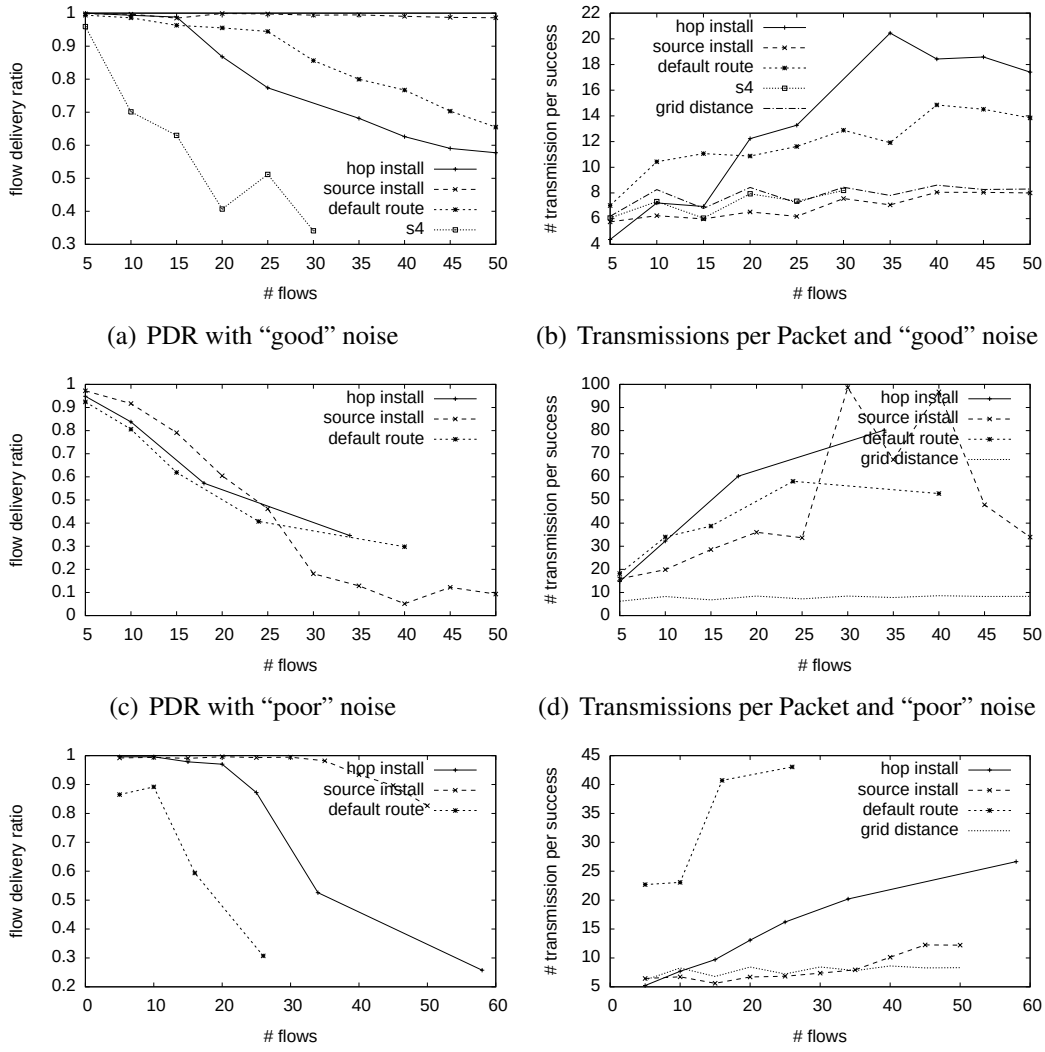


Figure 4.7: Our default route selection protocol compared with CTP in simulation with a poor noise model, and to results on our testbed.

In figure 4.8 we compare a number of design options with both the good and poor noise models. We begin by examining performance under the *good* noise model, varying



(a) PDR with "good" noise

(b) Transmissions per Packet and "good" noise

(c) PDR with "poor" noise

(d) Transmissions per Packet and "poor" noise

(e) PDR with "poor" noise and local traffic

(f) Transmissions per Packet and "poor" noise with local traffic

Figure 4.8: Simulation results from 225 Nodes arranged in a 15×15 Grid

the number of concurrent bidirectional flows from 5 to 50, at which point nearly 50% of the network is serving as a flow endpoint. Figure 4.8(a) examines the packet success rate, while figure 4.8(b) focuses on the number of transmissions per packet originated. Even with lossy links, HYDRO with Source-Install performs very well, maintaining a 98% delivery rate with 50 flows.

When using the good model, we also attempted to compare HYDRO to S4 by using the implementation the authors have released [51]. We were able to replicate their result of approximately 95% packet delivery with 5 concurrent flows, although admittedly with

somewhat different simulation parameters. We were confused by the sharp dropoff after this point; however, after examining the implementation we believe it can be explained by the lack of sufficient queuing in the packet forwarding component. We have observed that our own implementation can exhibit similar poor behavior when the queues are too short. Thus, we believe that S4 would perform much better at all traffic rates with a more sophisticated forwarding engine.

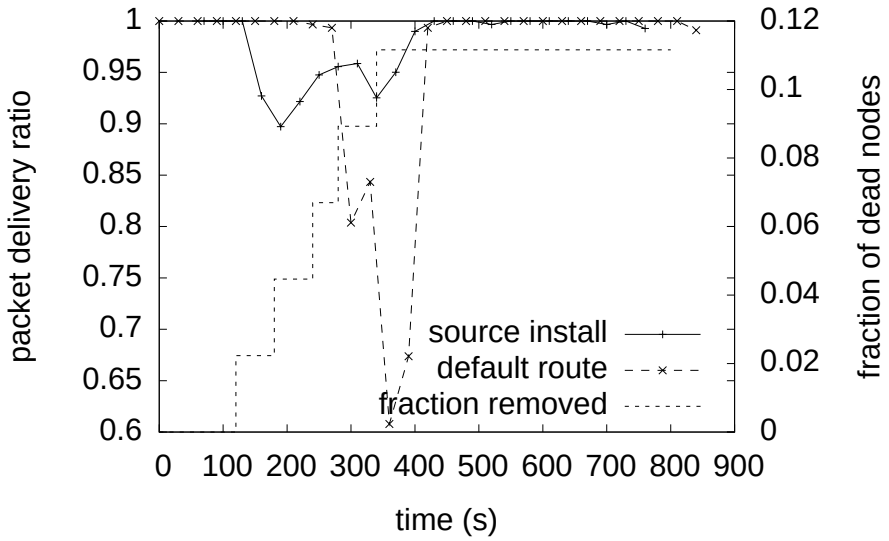
We also compare the number of transmissions necessary to deliver a packet in each of these situations. Notably, we perform similarly to S4 when installing routes which indicates we are collecting sufficient topology to build near-optimal routes.

For each of these tests, the flow table size was set to 25. What becomes clear once this is known is how hop-by-hop's performance deteriorates after intermediate tables cannot hold all the necessary state. Given that we are testing bidirectional flows, each flow requires two entries at intermediate routers. We can see that hop-by-hop and source install perform similarly up to 15 active flows, after which hop-by-hop deteriorates quickly. The reason for its deterioration is that install messages continue to be generated, even though tables have no room to accomodate them. While a more sophisticated controller policy could prevent this, the underlying limitation is fundamental to our design.

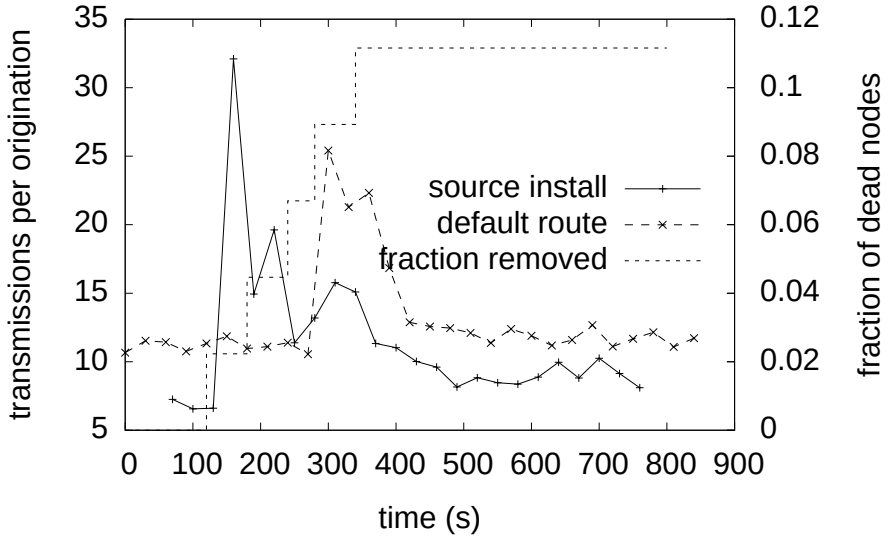
In figures 4.8(c) and 4.8(e), we show simulations with the poor noise model. Installed routes do not perform well in this regime, since the link Packet Reception Ratio is low enough so that there is a good probability that a packet will not be able to traverse a long path without failure; with this noise model, the PRR of the best links varies between 30% and 50%. Thus the protocol reverts to triangle routing in this regime, at the cost of both transmission stretch and delivery rate. However, we see in figure 4.8(e) that when traffic is restricted to a 5-hop neighborhood surrounding the source, the protocol performs much better. This reinforces our assertion that poor performance in 4.8(c) is due to path failure.

A final test shown in figure 4.9 attempts to evalute the protocol's ability to recover from failures. In this test, continuous flows of packets are started between 10 pairs of endpoints,

while other nodes are turned off at a rate of 5 every two minutes, until 10% of the network is dead. As paths are broken, the delivery rate drops and the transmission count increases as packets are re-routed through the controller and additional install messages are generated. The controller regains the topology relatively quickly and is able to repair the broken entries in the network and the transmission count drops again.



(a) Packet Delivery Rate



(b) Transmissions per Packet

Figure 4.9: 225 Nodes Arranged in a 15 x 15 Grid - 5 Nodes failing every two minutes, with 10 concurrent flows

Table 4.1: HYDRO State Requirements

	Hop	**Source**
Neighbor Table		
# of Entries	8	
Entry Size	22B	
Total Size	176B	
Routing Table		
# of Entries	6	
Entry Size	7B	7+(2*Hops)B
Max (Source)	42B	42+(12*Hops)B
Total (Path)	(7*Hops)B	7+(2*Hops)B
Max-7H(Source)	42B	126B
Total-7H(Path)	49B	21B

4.5 State and Control Overhead Analysis

The previous sections have primarily focused on the performance of HYDRO, namely stretch and packet delivery ratio. In this section we examine state requirements and control overhead, quantifying both in bytes and # of packets where appropriate.

4.5.1 State

The state stored at each node is comprised of two components: the neighbor-table, and the routing-table. Table 4.1 breaks down each of these. The size of a single neighbor-table entry is 22 bytes, and with a maximum size of 8 entries (in our current implementation), this results in 176 bytes per node for neighbor-table state.

The base size of a routing-table entry is 7 bytes, and for source install entries, the cost is an additional 2 bytes for each hop in the path. While the maximum state at a single node is greater with the source install option, solving the equation implies that the total state across the entire path for a given flow will be greater in the hop-by-hop case as long as the path length is greater than one. The size of the routing-table is limited to six entries in our implementation, although this parameter should be set to a realistic estimate of the

maximum number of destinations a node would be communicating with concurrently[2]. To provide sample numbers, we compute the total size across a node, and across a path when the path length is 7 hops, which was the average in the majority of our simulations.

4.5.2 Control Overhead

Table 4.2 details the control overhead of various HYDRO mechanisms, detailing the control overhead of each message, and the maximum and average (as observed in our evaluation) frequency of each mechanism. We note that the overhead only includes the portion specific to each mechanism; the cost of the entire message is not included as each of these can be piggybacked on data.

Topology Reports have a 4 Byte fixed cost, and then 2 Bytes per neighbor reported, which has empirically been less than half of the maximum 8 available. As a maximum they can be sent as fast as the periodic data rate, but in our experiments we generated them every 45 seconds, which was 1/22 of the data rate. Also, these figures are for the unoptimized case in which the full topology report is sent every period. Triggered and differential updates would reduce this overhead.

The full source route must be carried in each packet, which has a 4 Byte fixed cost, and then 2 Bytes per hop. If a packet is fragmented, the source route is only carried in the first fragment, and so this cost can be amortized over a full IPv6 packet, which is 11 6lowpan fragments.

Router solicitations only consume 1 bit, while router advertisements use 3 Bytes. Solicitations can theoretically be sent an infinite number of times if a default route is never found, but in practice only three are sent, the first to request advertisements, and the other to ensure advertisements weren't lost. Advertisements are triggered by solicitations, or changes in the default route, and so in the worst case can be equal to the neighborhood

[2]The need for hard-coded table sizes arises because of the lack of dynamic memory-allocation in our development environment; this is not a reflection of the protocol design.

density of a node. However, we use exponential timers to rate-limit advertisements, and since most nodes send solication messages almost concurrently, only one advertisement is sent in each of the three rounds of solications.

Table 4.2: HYDRO Control Overhead

		Frequency	
	Overhead	*Max*	*Avg*
Top. Report	4+(2*Neigh)B	Data Rate	45s
Src. Route	4+(2*Hops)B	Packet	Packet
Solicit.	1 Bit	Inf.	3
Adver.	3B	Density	3

4.5.3 Analysis

One key takeway is that neither state nor control overhead is dependent on the size of the network, and density only factors in to worst-case upper bounds. As such, HYDRO is able to remove many of the bottlenecks to scalability. To put these numbers into perspective, table 4.3 compares our results to published data for S4 [52]. As is always the case when results from different experimental methodologies are being compared, this comparison is simply meant to give a rough estimate of magnitude difference, rather then provide precise numerical differentiation.

In our worst case scenario, the total state at a given node would be 302 Bytes. On the other hand, the routing state in S4 grows as large as 1KB in some of their experiments, a significant portion of the 4KB of total device memory available. In terms of control traffic for setting up the network, lets assume that a node in HYDRO sends on topology report 7 hops to the controller (the average seen in our experiments), three solications, and three advertisements. This accounts for 96 Bytes of traffic for initial setup. Meanwhile, S4 reports setup costs of 500 Bytes per node in certain experiments.

In order to be fair, we point out that the per-packet overhead for data traffic in S4 is only 3 bytes, as opposed to our need to put the the full source route in the packet. Also, the S4

Table 4.3: Total Costs of HYDRO and S4 (Per Node)

	HYDRO	**S4**
Routing State	302B	1KB
Initial Control Traffic	96B	500B

results were from simulations with a larger network size than ours, although again our costs do not depend on network size. However, the network diameter affects the length of paths, and consequently the per-packet overhead for source routes, the number of hops topology reports must travel, and the size of route entries. We previously discussed optimizations to eliminate these factors, such as deploying multiple controllers, in section 3.3.

4.6 Revisiting the Core Challenges

We briefly examine how HYDRO performs in meeting the core challenges described in chapter 2.

4.6.1 Routing/Transmission Stretch

Transmission stretch is a key challenge in L2Ns because it must be minimized to conserve energy, yet the variability and inherently lossy nature of low-power radios make this difficult. HYDRO addresses this challenge by utilizing multiple link estimators that focus on accurately estimating the number of expected transmissions over a given link, and by aggregation, a given path, and continually refining and adjusting these link estimates. In addition, for point-to-point traffic, state is installed in the network to allow for routing over an optimized path.

4.6.2 Routing State

The resource-starved nature of typical L2N devices necessitate that careful attention be paid to the amount of routing state maintained at each node if the network is to be scalable.

We presented a state analysis in this chapter that provides absolute numbers and compares these to another protocol, S4 [52]. In addition, we highlighted in chapter 3 how our system meets the ROLL requirements that ensure manageable routing state.

4.6.3 Network Overhead

The typical low-data rate nature of L2N applications and the expensive energy cost of radio transmissions mandate control traffic be kept to a minimal, yet there is also a fundamental tension in maintaining an accurate view of a dynamic network. HYDRO attacks this problem by explicitly binding control traffic to data traffic by using data-driven triggers for link estimation and path installation. In addition, rather than attempt to maintain a complete picture of the global topology, HYDRO makes the tradeoff for reduced control traffic by maintaining a global view of *good* links in the network, which tend to be relatively stable, as we saw in our evaluation.

4.7 Summary

In this chapter we evaluated the HYDRO design presented in chapter 3 across a variety of platforms. We examined its performance on a real-world testbed, used a L2N-specific simulator to evaluate scalability properties, and finally used a simplified MATLAB model for basic sanity-checking and grounding of performance. In each of these cases, we focused on the reliability of HYDRO and its control overhead, evaluating multiple HYDRO designs in some cases. Subsequently we analytically examined the state and overhead requirements of HYDRO, and discussed how it met the core challenges presented in chapter 2.

Chapter 5

PLayer: A Policy-Aware Switching Layer for Data Centers

In previous chapters, we dicussed HYDRO, a routing protocol for low power networks that leveraged the inherent heterogenous two-tiered hierarchy. As another case study, in the next two chapters we examine the merits of two distinct design paradigms for PLayer, a switching layer for Datacenter Networks. PLayer was initially designed as a switching layer in which all information (routing policy and network topology), was pushed to individual switches, allowing for localized operation. Subsequently, PLayer was ported to NOX [33], an open sourced network control platform, in which the bulk of network intelligence are pushed to *network controllers*. We begin by focusing on the motivation for PLayer, and detail the design and evaluation of the initial distributed version in this chapter. In the next chapter we detail the centralizing effort, and highlight significant tradeoffs between the two design methodologies in this specific context.

5.1 Datacenter Routing and Infrastructure

In this section, we describe our target environment and the associated datacenter network architecture. We then illustrate the limitations of current best practices in datacenter mid-

dlebox deployment.

5.1.1 Datacenter Network Architecture

Our target network environment is characterized as follows:

Scale: The network may consist of tens (or hundreds) of thousands of machines running thousands of applications and services.

Middlebox-based Policies: The traffic needs to traverse various middleboxes, such as firewalls, intrusion prevention boxes, and load balancers before being delivered to applications and services.

Low-Latency Links: The network is composed of low-latency links which facilitate rapid information dissemination and allow for indirection-mechanisms with minimal performance overhead.

While both datacenters and many enterprise networks fit the above characterization, in this paper we focus on datacenters, for brevity.

The physical network topology in a datacenter is typically organized as a three layer hierarchy [12], as shown in Figure 5.1(a). The access layer provides physical connectivity to the servers in the datacenters, while the aggregation layer connects together access layer switches. Middleboxes are usually deployed at the aggregation layer to ensure that traffic traverses middleboxes before reaching datacenter applications and services. Multiple redundant links connect together pairs of switches at all layers, enabling high availability at the risk of forwarding loops. The access layer is implemented at the data link layer (*i.e.*, layer-2), as clustering, failover and virtual server movement protocols deployed in datacenters require layer-2 adjacency [1, 13].

5.1.2 Limitations of Existing Mechanisms

In today's datacenters, there is a strong coupling between the *physical* network topology and the *logical* topology. The logical topology determines the sequences of middleboxes

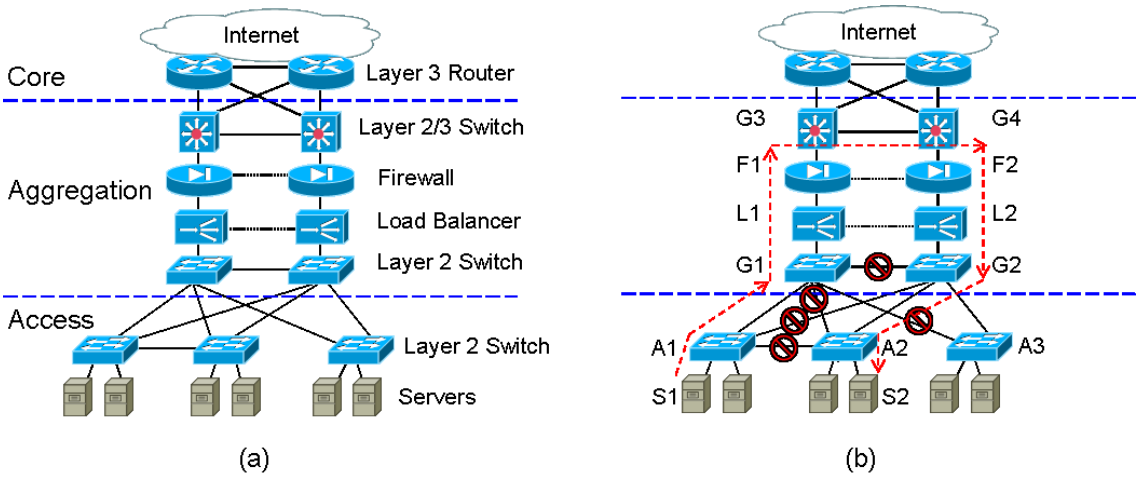


Figure 5.1: (a) Prevalent 3-layer datacenter network topology. (b) Layer-2 path between servers *S1* and *S2* including a firewall.

to be traversed by different types of application traffic, as specified by datacenter policies. Current middlebox deployment practices hard code these policies into the physical network topology by placing middleboxes in sequence on the physical network paths and by tweaking path selection mechanisms like spanning tree construction to send traffic through these paths. This coupling leads to middlebox deployments that are hard to configure and fail to achieve the three properties – correctness, flexibility and efficiency – described in the previous section. We illustrate these limitations using the datacenter network topology in Figure 5.1.

Hard to Configure and Ensure Correctness

Reliance on overloading path selection mechanisms to send traffic through middleboxes makes it hard to ensure that traffic traverses the correct sequence of middleboxes under all network conditions. Suppose we want traffic between servers *S1* and *S2* in Figure 5.1(b) to always traverse a firewall, so that *S1* and *S2* are protected from each other when one of them gets compromised. Currently, there are three ways to achieve this: (i) Use the existing aggregation layer firewalls, (ii) Deploy new standalone firewalls, or (iii) Incorporate firewall functionality into the switches themselves. All three options are hard to implement

and configure, as well as suffer from many limitations.

The first option of using the existing aggregation layer firewalls requires all traffic between *S1* and *S2* to traverse the path ($S1$, $A1$, $G1$, $L1$, $F1$, $G3$, $G4$, $F2$, $L2$, $G2$, $A2$, $S2$), marked in Figure 5.1(b). An immediately obvious problem with this approach is that it wastes resources by causing frames to gratuitously traverse two firewalls instead of one, and two load-balancers. An even more important problem is that there is no good mechanism to enforce this path between *S1* and *S2*. The following are three widely used mechanisms:

- *Remove physical connectivity:* By removing links $(A1, G2)$, $(A1, A2)$, $(G1, G2)$ and $(A2, G1)$, the network administrator can ensure that there is no physical layer-2 connectivity between *S1* and *S2* except via the desired path. The link $(A3, G1)$ must also be removed by the administrator or blocked out by the spanning tree protocol in order to break forwarding loops. The main drawback of this mechanism is that we lose the fault-tolerance property of the original topology, where traffic from/to *S1* can fail over to path $(G2, L2, F2, G4)$ when a middlebox or a switch on the primary path (*e.g.*, *L1* or *F1* or *G1*) fails. Identifying the subset of links to be removed from the large number of redundant links in a datacenter, while simultaneously satisfying different policies, fault-tolerance requirements, spanning tree convergence and middlebox failover configurations, is a very complex and possibly infeasible problem.

- *Manipulate link costs:* Instead of physically removing links, administrators can coerce the spanning tree construction algorithm to avoid these links by assigning them high link costs. This mechanism is hindered by the difficulty in predicting the behavior of the spanning tree construction algorithm across different failure conditions in a complex highly redundant network topology [22, 12]. Similar to identifying the subset of links to be removed, tweaking distributed link costs to simultaneously carve out the different layer-2 paths needed by different policy, fault-tolerance and traffic engineering requirements is hard, if not impossible.

- *Separate VLANs:* Placing *S1* and *S2* on separate VLANs that are inter-connected only at the aggregation-layer firewalls ensures that traffic between them always traverses a firewall. One immediate drawback of this mechanism is that it disallows applications, clustering protocols and virtual server mobility mechanisms requiring layer-2 adjacency [1, 13]. It also forces all applications on a server to traverse the same middlebox sequence, irrespective of policy. Guaranteeing middlebox traversal requires all desired middleboxes to be placed at all VLAN inter-connection points. Similar to the cases of removing links and manipulating link costs, overloading VLAN configuration to simultaneously satisfy many different middlebox traversal policies and traffic isolation (the original purpose of VLANs) requirements is hard.

The second option of using a standalone firewall to process *S1*-*S2* traffic is also implemented through the mechanisms described above, and hence suffers the same limitations. Firewall traversal can be guaranteed by placing firewalls on every possible network path between *S1* and *S2*. However, this incurs high hardware, power, configuration and management costs, and also increases the risk of traffic traversing undesired middleboxes. Packets traversing an undesired middlebox can hinder application functionality. For example, unforeseen routing changes in the Internet, external to the datacenter, may shift traffic to a backup datacenter ingress point with an on-path firewall that filters all non-web traffic, thus crippling other applications.

The third option of incorporating firewall functionality into switches is in line with the industry trend of consolidating more and more middlebox functionality into switches. Currently, only high-end switches [3] incorporate middlebox functionality and often replace the sequence of middleboxes and switches at the aggregation layer (for example, $F1$,$L1$,$G1$ and $G3$). This option suffers the same limitations as the first two, as it uses similar mechanisms to coerce *S1*-*S2* traffic through the high-end aggregation switches incorporating the required middlebox functionality. Sending *S1*-*S2* traffic through these switches even when a direct path exists further strains their resources (already oversubscribed by multiple ac-

cess layer switches). They also become concentrated points of failure. This problem goes away if all switches in the datacenter incorporate all the required middlebox functionality. Though not impossible, this is impractical from a cost (both hardware and management) and efficiency perspective.

Network Inflexibility

While datacenters are typically well-planned, changes are unavoidable. For example, to ensure compliance with future regulation like Sarbanes Oxley, new accounting middleboxes may be needed for email traffic. The dFence [49] DDOS attack mitigation middlebox is dynamically deployed on the path of external network traffic during DDOS attacks. New instances of middleboxes are also deployed to handle increased loads, a possibly more frequent event with the advent of on-demand instantiated virtual middleboxes.

Adding a new standalone middlebox, whether as part of a logical topology update or to reduce load on existing middleboxes, currently requires significant re-engineering and configuration changes, physical rewiring of the backup traffic path(s), shifting of traffic to this path, and finally rewiring the original path. Plugging in a new middlebox 'service' module into a single high-end switch is easier. However, it still involves significant re-engineering and configuration, especially if all middlebox expansion slots in the switch are filled up.

Network inflexibility also manifests as fate-sharing between middleboxes and traffic flow. All traffic on a particular network path is forced to traverse the same middlebox sequence, irrespective of policy requirements. Moreover, the failure of any middlebox instance on the physical path breaks the traffic flow on that path. This can be disastrous for the datacenter if no backup paths exist, especially when availability is more important than middlebox traversal.

Inefficient Resource Usage

Ideally, traffic should only traverse the required middleboxes, and be load balanced across multiple instances of the same middlebox type, if available. However, configuration inflexibility and on-path middlebox placement make it difficult to achieve these goals using existing middlebox deployment mechanisms. Suppose, spanning tree construction blocks out the $(G4, F2, L2, G2)$ path in Figure 5.1(b). All traffic entering the datacenter, irrespective of policy, flows through the remaining path $(G3, F1, L1, G1)$, forcing middleboxes $F1$ and $L1$ to process unnecessary traffic and waste their resources. Moreover, middleboxes $F2$ and $L2$ on the blocked out path remain unutilized even when $F1$ and $L1$ are struggling with overload.

5.2 PLayer Design Overview

5.2.1 PLayer Goals

The policy-aware switching layer (PLayer) is a datacenter middlebox deployment proposal that aims to address the limitations of current approaches, described in the previous section. The PLayer achieves its goals by adhering to the following two design principles:

(i) Separating policy from reachability. The sequence of middleboxes traversed by application traffic is explicitly dictated by datacenter policy and not implicitly by network path selection mechanisms like layer-2 spanning tree construction and layer-3 routing.

(ii) Taking middleboxes off the physical network path. Rather than placing middleboxes on the physical network path at choke points in the network, middleboxes are plugged in off the physical network data path and traffic is explicitly forwarded to them. Explicitly redirecting traffic through off-path middleboxes is based on the well-known principle of

indirection [66, 69, 31]. A datacenter network is a more apt environment for indirection than the wide area Internet due to its very low inter-node latencies.

5.2.2 Policy-Aware Switches

The PLayer consists of enhanced layer-2 switches called policy-aware switches or *pswitches*. Unmodified middleboxes are plugged into a *pswitch* just like servers are plugged into a regular layer-2 switch. However, unlike regular layer-2 switches, *pswitches* forward frames according to the policies specified by the network administrator.

5.2.3 Policy Specification

Policies define the sequence of middleboxes to be traversed by different traffic. A policy is of the form: *[Start Location, Traffic Selector]→Sequence*. The left hand side defines the applicable traffic – frames with 5-tuples (*i.e.*, source and destination IP addresses and port numbers, and protocol type) matching the *Traffic Selector* arriving from the *Start Location*. We use *frame 5-tuple* to refer to the 5-tuple of the packet within the frame. The right hand side specifies the sequence of middlebox types (not instances) to be traversed by this traffic [1].

Policies are automatically translated by the PLayer into *rules* that are stored at *pswitches* in rule tables. A rule is of the form *[Previous Hop, Traffic Selector] : Next Hop*. Each rule determines the middlebox or server to which traffic of a particular type, arriving from the specified previous hop, should be forwarded next. Upon receiving a frame, the *pswitch* matches it to a rule in its table, if any, and then forwards it to the next hop specified by the matching rule.

[1] Middlebox interface information can also be incorporated into a policy. For example, frames from an external client to an internal server must enter a firewall via its *red* interface, while frames in the reverse direction should enter through the *green* interface.

5.2.4 Centralized Components

The PLayer relies on centralized *policy* and *middlebox* controllers to set up and maintain the rule tables at the various *pswitches*. Network administrators specify policies at the policy controller, which then reliably disseminates them to each *pswitch*. The centralized middlebox controller monitors the liveness of middleboxes and informs *pswitches* about the addition or failure of middleboxes.

5.3 PLayer Performance

5.3.1 Implementation

We have prototyped *pswitches* in software using Click [43] (kernel mode). An unmodified Click *Etherswitch* element formed the Switch Core. The Click elements representing the Policy Core were implemented in 5500 lines of C++. Each port of the Policy Core plugs into the corresponding port of the Switch Core, thus satisfying the separation between the Policy Core and the Switch Core to facilitate reuse of existing functionality.

Due to our inability to procure expensive hardware middleboxes for testing, we used commercial quality software middleboxes running on standard Linux PCs: (i) an *iptables* [11] based firewall, (ii) a Bro [57] intrusion detection system, and (iii) a *BalanceNG* [2] load balancer. We used the Net-SNMP [5] package for implementing SNMP-based middlebox liveness tracking. *snmpd* daemons running on the middlebox PCs send SNMP traps to the *snmptrapd* daemon running on the PC running our prototype middlebox controller implemented in Ruby On Rails [10]. The rapid prototyping features of Ruby On Rails were leveraged to prototype the policy controller and the web-based policy configuration GUI.

5.3.2 Preliminary Evaluation Results

In this section, we provide preliminary throughput and latency benchmarks for our prototype *pswitch* implementation, relative to standard software Ethernet switches and on-path middlebox deployment. Our initial implementation focused on feasibility and functionality, rather than optimized performance. While the performance of a software *pswitch* may be improved by code optimization, achieving line speeds is unlikely. Inspired by the 50x speedup obtained when moving from a software to hardware switch prototype with Ethane [17], we plan to prototype *pswitches* on the NetFPGA [6] boards. We believe that the hardware *pswitch* implementation will have sufficient switching bandwidth to support frames traversing the *pswitch* multiple times due to middleboxes and will be able to operate at line speeds.

Our prototype *pswitch* achieved 82% of the TCP throughput of a regular software Ethernet switch, with a 16% increase in latency. Figure 5.2(a) shows the simple topology used in this comparison experiment, with each component instantiated on a separate 3GHz Linux PC. We used *nuttcp* [8] and *ping* for measuring TCP throughput and latency, respectively. The *pswitch* and the standalone Click Etherswitch, devoid of any *pswitch* functionality, saturated their PC CPUs at throughputs of 750 Mbps and 912 Mbps, respectively, incurring latencies of 0.3 ms and 0.25 ms.

Compared to an on-path middlebox deployment, off-path deployment using our prototype *pswitch* achieved 40% of the throughput at double the latency (Figure 5.2(b)). The on-path firewall deployment achieved an end-to-end throughput of 932 Mbps and a latency of 0.3 ms, while the *pswitch*-based firewall deployment achieved 350 Mbps with a latency of 0.6 ms. Although latency doubled as a result of multiple *pswitch* traversals, the sub-millisecond latency increase is in general much smaller than wide-area Internet latencies. The throughput decrease is a result of packets traversing the *pswitch* CPU twice, although they arrived on different *pswitch* ports. Hardware-based *pswitches* with multi-gigabit switching fabrics should not suffer this throughput drop.

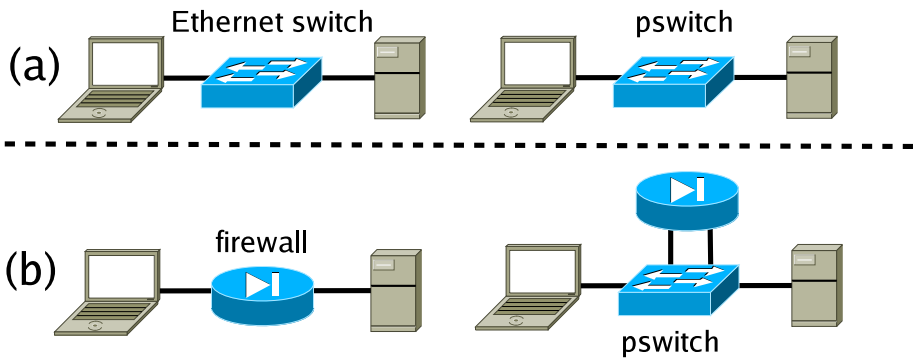


Figure 5.2: Topologies used in benchmarking *pswitch* performance.

Microbenchmarking showed that a *pswitch* takes between 1300 and 7000 CPU ticks (1 tick $\approx \frac{1}{3000}$ microsecond on a 3GHz CPU) to process a frame, based on its destination. A frame entering a *pswitch* input port from a middlebox or server is processed and emitted out of the appropriate *pswitch* output ports in 6997 CPU ticks. Approximately 50% of the time is spent in rule lookup (from a 25 policy database) and middlebox instance selection, and 44% on frame encapsulation. Overheads of packet classification and packet handoff between different Click elements consumed the remaining processing time. An encapsulated frame reaching the *pswitch* directly attached to its destination server/middlebox was decapsulated and emitted out to the server/middlebox in 1312 CPU ticks.

5.4 Summary

In this chapter we began by describing the extensive use of middleboxes in datacenters, and the challenges of deploying them today, despite a host of potential mechanisms. Subsequently, we focus on some of the key principles that make this deployment difficult at a higher level, which must be addressed by any solution. We then present PLayer, a policy-aware switching layer for datacenters that elevates middleboxes to first-class citizens in a network. We discuss the main goals for PLayer, and then describe the functionality of its various components, from policy specification to switch-based policy execution. Finally, we describe our implementation of PLayer and preliminary evaluation results. Many details

were omitted in this chapter in order to focus on the necessary groundwork, preparing for our discussion of a centralized PLayer design in chapter 6; a much more thorough overview is provided by Joseph et al. [41], including a more extensive evaluation section with indepth functionality validation.

Chapter 6

Centralizing PLayer

In the previous chapter we introduced PLayer, a policy-aware switching layer for Data-centers. With the exception of a few central components, by design the implementation was primarily decentralized, both in the distribution of state, as well as decision making capabilities.

In this chapter, we examine the process of converting PLayer into a centralized protocol. We use NOX [33], a centralized open-source control platform for networking as the foundation, and port the principle concepts of PLayer onto this system.

We begin by providing a brief overview of NOX, followed by a description of the centralized PLayer implementation and preliminary evaluation results. We conclude by comparing the two implementations across a set of (mostly qualitative) benchmarks.

6.1 NOX: A Centralized Control Platform for Networking

Based primarily on Ethane [17], NOX is primarily designed to simplify the creation of software for controlling and monitoring networks [7]. It has been primarily designed for large enterprise networks with multiple switches and thousands of hosts. At its core, NOX is providing access to network state, such as topology and network host information, and full communication connectivity access. Developers have access to how the network performs

forwarding and routing, with the ability to operate at the flow level. In addition, access to higher-level primitives, such as user and host access and policies are provided. A base set of applications, such as topology management, shortest path routing, and user/host access authentication and policies are bundled with NOX.

6.1.1 Architectural Overview

The NOX architecture includes four components: the centralized NOX controller (§ 6.1.3), OpenFlow Programmable Switches (§ 6.1.2), end hosts, and users. The controller is a single logical entity, although it can be replicated for scalability and fault tolerance. Each switch notifies the controller of its existence, and creates a secure direct channel to it for control traffic.

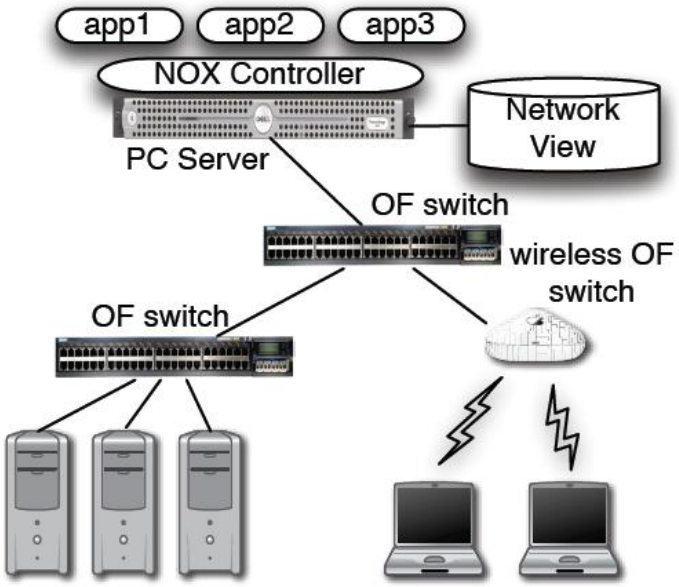


Figure 6.1: Example NOX Network Architecture

Whenever an end host connects to a switch, the switch reports the presence of the host, and the port it is attached to, to the controller, enabling the building of a complete topological view of the network. Periodic beacons and standard failure detection mechanisms enable the controller to maintain a consistent global view of the network. When a user joins

the network, it registers with the controller, notifying it which host(s) it is associated with.

Network administrators can specify policies at the NOX controller. Many policies are security based (which isn't surprising, given Ethane's, and its predecessor Sane's [16], network security based roots). Users and hosts can be placed in groups, and then access restrictions can be created, such as creating a virtual partition between two departments in an enterprise.

6.1.2 OpenFlow Programmable Software Switches

Routers and switches have traditionally been predominantly hardware based. While the proponents of programmable routers point to additional control, flexibility, rapid-prototyping capabilities, ultimately the main determining factor has been performance. Switches/Routers must be able to function at line speeds, and software routers, such as Click [43] have been unable to match the performance of switching fabrics and TCAMs.

OpenFlow switches [54] provide an intermediary balance, providing both hardware and software functionalities with the separation focused on the flow table. A flow table entry is composed of two parts: the packet 10-tuple for classification, and the appropriate action to be taken. The 10 fields specified are as follows:

- Ingress Port
- Ethernet Source and Destination Addresses, and Type
- IP Source and Destination Addresses, and IP Protocol
- VLAN ID
- TCP Source and Destination Ports

A flow-table entry can have all these fields specified, or use the *ANY* value to symbolize a *don't care*. When a packet arrives, the switch performs a hardware lookup to attempt to

classify the packet according to its 10-tuple. If a matching flow-table entry is found, the switch performs the action dictated by the entry. These include forwarding the packet (including how to forward it, e.g. through which port), as well as providing the opportunity to modify the packet's 10-tuple fields (e.g. change Ethernet/IP source/destination addresses).

Flow table entries can be naturally populated by traditional means, such as an L2 spanning tree algorithm and address learning. Alternatively, OpenFlow exports a software interface to also enable external sources to manipulate entries; they can be inserted, deleted and modified.

OpenFlow switches come in three different versions: there is linux software for turning a PC into an OpenFlow switch, a netFPGA [6] build which provides 4 ports, and various OpenFlow enabled commercial switches.

6.1.3 NOX Centralized Controller

NOX is an open-source centralized network management platform. We discussed the OpenFlow-enabled switches previously, but these simply provide an interface that allows for remote control of switches. The NOX Centralized Controller maintains the bulk of the intelligence, and serves as the point of interaction with administrators of the network.

The NOX Controller (NOX from here on out) functions as a layered event-driven system. Events can be generated by any component in the system, whether the joining of a new server, failing of a link on a switch, or software components within NOX themselves. Each event is only processed by a single component at any given time, and each component has the option to allow others to subsequently process the event or to end processing. A user-specified ordering dictates the processing order of an event by components. In addition, derivative events provide a loose version of layering and service interfaces. For example, a low-level NOX component receives a *packet-in* event, signifying a packet has been received from a switch, and upon examination, fires off a *flow-in* event, allowing for higher-level components to operate on this new event. These derivative events typically

have additional or modified semantics as a result of processing triggered by the original event. In such a model, the higher-level component is agnostic to who fires off the event, or the previous processing completed, as long as the data provided by the event meets the expected definition maintained by the higher level component.

A set of common components comes bundled with NOX, two of which are particularly useful for our needs. A *Topology* component maintains a global view of the network topology, as well as end-user and host mappings. In addition, a *Routing* component computes shortest-path routes between any two entities in the network, and uses event notifications from *Topology* to update these routes.

6.2 Centralized PLayer Implementation Over NOX

In the last chapter we discussed the design of the original PLayer, which was predominantly distributed and made use of modified policy-aware switches, or *pswitches*. We now focus on how to push the intelligence of the network to the edge, centralizing it at a NOX controller and maintain only barebones capabilities at the switches themselves.

6.2.1 Design Overview

In centralizing PLayer, we were able to make use of many of the tools provided by NOX, significantly easing our implementation task.

In the distributed case, each *pswitch* maintains a full policy-table at each switch, and when packets arrive, it classifies each packet, and then determines where in the sequence the packet is. It stores decisions in a *Rule Table*, which can be used to facilitate future decisions. PLayer does not require any network topology information as it simply uses existing layer-2 mechanisms for forwarding packets. It makes use of encapsulation (and baby giant packet formats) to directly address next-hop middleboxes in a policy sequence.

Using NOX, the design undergoes some obvious shifts, primarily that the bulk of the

intelligence is moved into the centralized controller, and the switches only execute forwarding according to installed flow entries. To explain the architecture of our design, we walk through the process for a new flow created in the network:

1. **Classification:** When a new packet arrives at a switch, the switch begins by checking its flow table for a match. If none is found, the switch forwards the packet to a controller, triggering a *packet-in* event. The *flow-in* event, a derivative event, is passed up the chain until it reaches our *PLayer Core Component*. The *PLayer Core Component* checks its specified policies list to find a match. If none is found, the new flow is ignored.[1]

2. **Instance Selection:** If a match is found during the *Classification* step, then the accompanying traversal sequence is also obtained. This sequence only indicates a *type* of middlebox, rather than a specific instance. At this point, the *PLayer Core Component* consults the *PLayer Topology Component*, which provides a mapping between a type and an actual instance. In the current implementation, the *PLayer Topology Component* uses a round-robin hashing scheme to select a particular instance.

3. **Route Setup:** After obtaining a specific instance for each middlebox in the sequence, the *PLayer Core Component* generates an entire path for packets in the flow by making repeated calls to the *Routing* component, which returns the shortest path between any two entities. Once the path has been generated, the OpenFlow interface is used to install the appropriate flow entries at all switches along the path, completing the process.

[1]NOX actually provides a classification engine in which a component can specify that it only wants to receive *flow-in* events when the new flow matches a certain combination (e.g. a policy specification). We use this to only receive pertinent *flow-in* events.

Edge Cases

The steps discussed above handle the general case of middlebox traversal, but there are a set of deviations, some more common than others, that our design must consider. In many situations, we are able to use solutions similar to that used in the original PLayer work:

- **Ambiguous Previous Hop:** At any given point, the action a switch must take when receiving a packet is determined by a policy-specified traversal sequence, and so it is critical to know how much of the sequence has been traversed at any given point. The original PLayer used encapsulation to explicitly identify the previous hop and the next-hop destination, removing any such ambiguation. The centralized design uses flow entries to explicitly setup the path without using encapsulation (which is not currently provided by OpenFlow regardless). The flow entries use the incoming port and the packet 10-tuple for classification, which becomes problematic if that combination is not unique during a traversal sequence, as is the case in the example demonstrated in figure 6.2. If we assume that the policy specifies that the middleboxes are to be traversed numerically, when switch 2 receives the packet, it is not sure whether to send it to MB_2 or the destination, since it does not know if the packet just came from the MB_1 or MB_3. In such a case, an additional classifier is needed, such as potentially using the VLAN tag, and modifying the tag during the traversal in order for this new flow entry to be unique.

- **Middlebox Addressing:** In some cases, middleboxes have specific requirements, such as needing a packet to be addressed to them at layer-2, otherwise the packet is dropped. Such information can be recorded in the *PLayer Topology Component*, and flow entries for the connected switch altered so that the layer-2 address is overwritten and reverted before and after the middlebox traversal, respectively.

- **Non-Transparent Middleboxes:** Some middleboxes alter elements in the packet 10-tuple, making a consistent flow entry for the entire traversal path infeasible. For

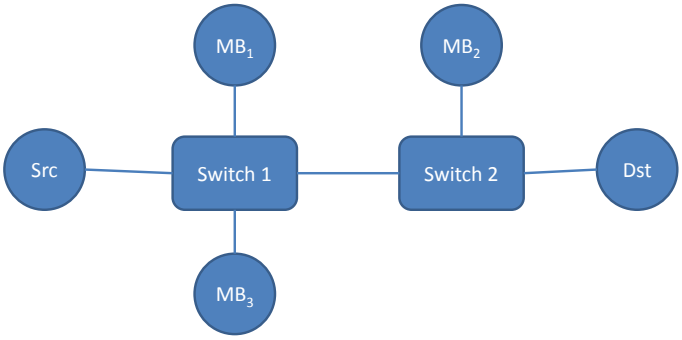


Figure 6.2: An example of an ambiguous previous hop during policy traversal, assuming middleboxes are to be traversed in numeric order.

example, load balancers will often insert their own IP address in the source-IP field, and the IP address of the selected server in the destination-IP field. In other cases, packets will be multiplexed, or decrypted (SSL offload boxes). Similar to the original PLayer design, the centralized version uses per-segment policies and corresponding flow installations, where a segment is defined as a portion of a path during which modifications occur at the end-points. If a modification is deterministic, the appropriate flow entry for the segment can be installed in response to the original *flow-in* event. However, in cases of non-determinism, such as a load balancer selecting a web-server, this final segment can be treated as a separate flow without loss of functionality. The only requirement is that packets traversing the path in the reverse direction use the same middlebox instance, which can be guaranteed by installing bidirectional flow entries initially.

6.2.2 Initial Evaluation Results

The bulk of our evaluation is qualitative, as it focused on ease of design. We provided performance results for the original PLayer in chapter 5. Even in that case, the focus was primarily on functionality, as using a PC-based router where CPU saturation was an issue provided skewed results, relative to hardware-based solutions. With the centralized version, for performance results we essentially rely on those of NOX, which has demonstrated

the ability to handle a large amount of new flows with minimal flow install latency. Additional processing at NOX does increase flow latency but we have found the increase to be relatively small and have yet to optomize it. In addition, this cost is only incurred by the first packet in a flow.

Our reliance on NOX's performance numbers highlight a fundamental point of our centralized design: ease of implementation. For the sake of discussion, we don't differentiate between the infrastructure of NOX and the notion of a more abstract centralized paradigm, as we feel NOX provides a faithful embodiment of a centralized general network management platform.

When implementing the centralized PLayer, the process was greatly facilitated by the primitives and infrastructure provided by NOX. NOX handles communication with switches in the network, and maintains a global topology view. It also has a default routing engine that can calculate shortest path routes between any two destinations in the network. With these in place, there was only a minimal set of components (which we discussed above), that we were required to build. NOX certainly was not designed with PLayer in mind, but this seems to validate the notion that a centralized design helps modularize functionality and allow for increased reusability rather than redundant implementations.

6.2.3 Extensions and Future Work

Up until this point, the focus has been on precisely replicating the capabilities of the original PLayer design. However, there are numerous extensions and additions that can be considered, and we discuss two of these here:

Load Balancing

The current design uses a round-robin scheme to select a middlebox instance and simple shortest-path routing to find paths. Much more complex algorithms can be used for each. First, by instituting a mechanism for polling middlebox instances, their actual load could be

ascertained, and hence when selecting an instance, the least loaded one could be used. This would however introduce additional complexity, because while round-robin hashing was deterministic, load-based selection is not, and so flow state would need to be maintained to ensure that all packets in a flow (in both directions) traverse the same middlebox instance. This is equivalent to using a third-party load balancer (either software or hardware) in conjunction with NOX.

Second, load balancing across paths could be used with a more complex routing algorithm. For example, an ECMP-like algorithm could be used to spread the load across all the paths to a given middlebox instance. While this load-balancing algorithm could be developed as part of PLayer, most likely load-balancing would be an integral part of the centralized infrastructure, as discussed by Tavakoli et al. [67], and PLayer could be built on top of it, oblivious to the underlying path selection algorithm.

Policy Automation and Middlebox Modeling

The current implementation of the centralized PLayer (and the original PLayer as well) primarily focuses on transparent middleboxes with no special needs, such as proper layer-2 addressing, etc., and in the presence of such needs, resorts to manually specified modifications. However, such an approach is cumbersome, as well as impractical, when scaling to large datacenter networks with thousands of middleboxes. Instead, a mechanism is needed for automating such tasks, which needs to be provided in two steps.

We briefly referred to the first step earlier. In the *PLayer Topology Component*, information about each middlebox should be recorded. This includes which fields in the header are modified and in what fashion by the middlebox, and also what form of addressing is needed (e.g. does the packet have to be addressed to the middlebox at layer-2?). When the *PLayer Core Component* asks the *PLayer Topology Component* to select instances of the middleboxes, the *PLayer Topology Component* would also return this metadata, allowing the *PLayer Core Component* to modify the flow table entries appropriately.

The second step in this process is obtaining this metadata about each of these middleboxes. Joseph et al. [40] describe a middlebox modeling scheme in which the functionality of middleboxes is decomposed and a series of processing decisions across multiple packets is examined to determine the precise characteristics of a given middlebox instance. This information can then be fed into the *PLayer Topology Component*.

6.3 Comparison of Both Implementation Paradigms

We have now outlined the design of both the original, distributed PLayer, as well as a newer centralized version built on top of NOX. Quantitative performance results are not particularly comparable, as the centralized version uses an established platform with hardware forwarding provided at line rates, while the original version uses an unoptimized PC-based forwarding engine. The only aspect of performance under consideration is flow-install latency, and this has been shown to be acceptable under NOX, and should only be incurred by a single packet in a given flow.

The main point of interest for comparison is ease of implementation, and in a broader context, flexibility. When developing the original PLayer, most functionality had to be created from scratch, and despite efforts to maintain good software design principles, inevitability the design shifted towards a monolithic stack. One of the main reasons for this is that different applications have varying functionality and state needs and characteristics, and hence are difficult to design completely modularly in a distributed fashion. This makes two components difficult: changes to existing policy (such as routing policy), and interoperability with other components of a datacenter networking architecture.

On the other hand, the centralized version significantly simplifies such tasks. By decomposing the system across two axes, functionality and state, it enables a level of modularity for developing individual components without affecting the rest of the system. For example, as discussed earlier, the policy for selecting routes between any two components,

as well as various load balancing policies, could all be easily incorporated within the current design. In addition, as laid out by Tavakoli et al. [67], the PLayer scheme cleanly integrates with a host of other mechanisms for addressing the needs of the datacenter.

6.4 Summary

In this chapter we presented a centralized design of PLayer, the policy-aware switching layer for datacenters that was introduced in chapter 5. We began by describing a centralized network architecture, revolving around a centralized NOX controller and programmable OpenFlow switches, as these form the basis for our design. Subsequently, we present the design of a centralized PLayer, building on top of this NOX-based architecture. We walked through an example of flow-processing to demonstrate the design, and then highlighted edge cases that must be addressed, and future extensions to our work. From an evaluation perspective, we were able to rely on the results provided by NOX itself, and so the main focus became ease of implementation, which was significantly greater than the original PLayer due to extensive reuse that we were able to leverage. The comparison of the two different design paradigms focused mainly on this ease of implementation aspect, and also on the flexibility to integrate with solutions to address other challenges in the datacenter, as outlined by Tavakoli et al. [67].

Chapter 7

Conclusion

7.1 Contribution Summary

This dissertation consisted of two main contributions to address the challenges of routing in Lossy and Low-Powered Networks, and Large-scale Datacenters.

We presented HYDRO, a routing protocol for L2Ns that begins with the premise that centralized triangle routing provides any-to-any routing with constant state and minimum complexity at the cost of moderate stretch. Exploring additional design points in the centralized space yielded minimum stretch with an acceptable amount of space. The aim of this dissertation was not to present HYDRO or centralized routing as the panacea for L2N routing. Rather, this dissertation explored the challenges of designing centralized solutions for L2Ns, selected a specific point within the design space, and demonstrated its validity in experiments under varying conditions. The goal is to encourage exploration of additional design points in the centralized space, with various capabilities and tradeoffs, coupled with an indepth performance evaluation in a representative, yet exhaustive fashion.

Next we discussed PLayer as a system to address one of the core challenges of datacenter networking: middleboxes. Despite being-treated as second-class network citizens in the Internet, middleboxes have become an indispensable part of routing within today's

architecture, particularly in large datacenter networks. In order to facilitate deployments of these middleboxes, we presented PLayer, a policy-aware switching layer for datacenters that enables middleboxes to be taken off the physical path, and be explicitly traversed according to network policy specifications. Subsequently, in keeping with the central hypothesis of this dissertation, we explored a design for centralizing PLayer, and discussed the tradeoffs relative to the original PLayer.

7.2 Analysis and Discussion of Future Roadmap

Both of these pieces of work have mainly focused on demonstrating the viability of a centralized approach. Although undoubtedly there is more work to be done in testing the resilience and performance of these solutions, much future work lies in exploring the flexibility and control that is gained with a centralized solution.

With regards to HYDRO, the current version uses a very simplied global topology view and route installation policy. Much more complex routing metrics can be used, such as routing based on energy, or wanting to avoid hotspots in terms of state maintained by any particular node. Also, in many cases traffic patterns are highly predictable in L2N applications, and so route installation policies can be tailored to the traffic model (e.g. only installing routes for long flows, or preloading routes to avoid unnecessary controller traffic). To test the flexibility of this centralized model, it would be interesting to go through the exercise of emulating existing protocols and algorithms to understand where the limitations of this flexibility lie. Finally, different mechanisms have to be explored for dealing with mobile nodes, such as having them use different beaconing rates, and modified link estimation techniques that form the basis for neighbor discovery.

We discussed some extensions for a centralized PLayer system, such as automating the deployment of a middlebox using modeling and policy automation. However, the bulk of research will likely be in putting this work in the context of the larger set of datacenter

requirements, and exploring what such a solution would look like. Tavakoli et al. [67] start down this road, putting forth NOX as a general network management platform, capable of addressing datacenter needs, and integrate PLayer into this work.

As a final point for discussion, we return to our original hypothesis, that a hybrid routing protocol which combines centralized control and local flexibility is best suited for these environments, and ask: how well did it hold?

With HYDRO, there is definitely more work to be done in examining the affects of shifting functionality between the two operational paradigms, but the high-level takeaway was that this combination worked well and overcame the limitations of each when deployed individually. The centralized aspect provides the ability to enact precise routing policies with global knowledge, and control the tradeoff between state and stretch. At the same time, the local flexibility was invaluable in providing reliability in the face of an inherently dynamic and lossy environment.

In the datacenter environment, the answer still needs to be flushed out more. At a high level, local functionality means less precise central control (and often visibility/transparency), but in many cases may be needed to address latency and reliability concerns. It is clear that centralized control is a key component, and the scale and complexity of many datacenters make purely distributed solutions unmanageable. However, centralization does have its limits, as constant communication with a centralized controller is untenable, and flow setup latencies can potentially be larger than entire flow durations. One way to deal with these challenges is to operate the centralized system proactively, rather than reactively, pushing the operational instructions pre-emptively. However, this is mainly effective in scenarios where the workload can be predicted ahead of time. In reality, there are certain local primitives that greatly facilitate network operations. As examples, ECMP allows for load balancing without per-flow state or per-flow communication with the controller, and local recovery mechanisms help prevent packet losses while failure notifications propagate to the controller, and new paths are computed and installed. A key aspect of future re-

search will be understanding what the fundamentally necessary local primitives are and determining where the boundary between centralized control and local agility must fall.

Bibliography

[1] Architecture Brief: Using Cisco Catalyst 6500 and Cisco Nexus 7000 Series Switching Technology in Data Center Networks. http://www.cisco.com/en/US/prod/collateral/switches/ps9441/ps9402/ps9512/White_Paper_C17-449427.pdf.

[2] BalanceNG: The Software Load Balancer. http://www.inlab.de/balanceng.

[3] Cisco Catalyst 6500 Series Switches Solution. http://www.cisco.com/en/US/products/sw/iosswrel/ps1830/products_feature_guide09186a008008790d.html.

[4] Mobile ad-hoc networks (manet). http://www.ietf.org/html.charters/manet-charter.html.

[5] Net-SNMP. http://net-snmp.sourceforge.net.

[6] NetFPGA. http://netfpga.org.

[7] Nox: An openflow controller. http://www.noxrepo.org.

[8] nuttcp. http://linux.die.net/man/8/nuttcp.

[9] Routing - wikipedia. http://en.wikipedia.org/wiki/Routing.

[10] Ruby on Rails. http://www.rubyonrails.org.

[11] The netfilter.org project. `http://netfilter.org`.

[12] Cisco Data Center Infrastructure 2.1 Design Guide, 2006.

[13] Mauricio Arregoces and Maurizio Portolani. *Data Center Fundamentals*. Cisco Press, 2003.

[14] A. Brandt and G. Porcu. Home automation routing requirements in low power and lossy networks. In *IETF Internet Draft*, 2009.

[15] Matthew Caesar, Donald Caldwell, Nick Feamster, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. Design and implementation of a routing control platform. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 15–28, Berkeley, CA, USA, 2005. USENIX Association.

[16] Martin Casado, Tal Garfinkel, Aditya Akella, Michael Freedmanand Dan Boneh, Nick McKeown, and Scott Shenker. SANE: A Protection Architecture for Enterprise Networks. In *Usenix Security*, 2006.

[17] Martn Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, and Nick Mckeown. Ethane: Taking control of the enterprise. In *SIGCOMM Computer Comm. Rev*, 2007.

[18] Chen, J. and Karric Kwong and Chang, D. and Luk, J. and Bajcsy, R. Wearable Sensors for Reliable Fall Detection. *Engineering in Medicine and Biology Society, 2005. IEEE-EMBS 2005. 27th Annual International Conference of the*, pages 3551 – 3554, Jan. 2005.

[19] T. Clausen and P. Jacquet. RFC 3626: Optimized link state routing protocol, 2003.

[20] M. Dohler, T. Watteyne, T. Winter, and D. Barthel. Urban wsns routing requirements in low power and lossy networks. In *IETF Internet Draft*, 2009.

[21] Prabal Dutta, Mark Feldmeier, Joseph Paradiso, and David Culler. Energy metering for free: Augmenting switching regulators for real-time monitoring. In *IPSN '08: Proceedings of the 7th international conference on Information processing in sensor networks*, pages 283–294, Washington, DC, USA, 2008. IEEE Computer Society.

[22] K. Elmeleegy, A.L. Cox, and T.S.E Ng. On Count-to-Infinity Induced Forwarding Loops Ethernet Networks. In *Infocom 2006*.

[23] Nick Feamster, Hari Balakrishnan, Jennifer Rexford, Aman Shaikh, and Kobus van der Merwe. The Case for Separating Routing from Routers. In *ACM SIGCOMM Workshop on Future Directions in Network Architecture (FDNA)*, Portland, OR, September 2004.

[24] Rodrigo Fonseca, Omprakash Gnawali, Kyle Jamieson, and Philip Levis. Collection tree protocol. `http://www.tinyos.net/tinyos-2.x/doc/html/tep119.html`.

[25] Rodrigo Fonseca, Omprakash Gnawali, Kyle Jamieson, and Philip Levis. Four-bit wireless link estimation. In *Sixth Workshop on Hot Topics in Networks (HotNets VI)*, 2007.

[26] Rodrigo Fonseca, Sylvia Ratnasamy, Jerry Zhao, , Cheng Tien Ee, David Culler, Scott Shenker, and Ion Stoica. Beacon vector routing: Scalable point-to-point routing in wireless sensornets. In *Proceedings of the Second USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI 2005)*, 2005.

[27] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: Global comprehension for distributed replay. In *NSDI: Proceedings of the annual conference on Networked Systems Design and Implementation*, 2007.

[28] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay debugging for distributed applications. In *ATEC '06: Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, 2006.

[29] Omprakash Gnawali, Rodrigo Fonseca, Kyle Jamieson, David Moss, and Philip Levis. Collection tree protocol. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2009.

[30] Omprakash Gnawali, Ben Greenstein, Ki-Young Jang, August Joki, Jeongyeup Paek, Marcos Vieira, Deborah Estrin, Ramesh Govindan, and Eddie Kohler. The tenet architecture for tiered sensor networks. *ACM Sensys*, 2006.

[31] Richard Gold, Per Gunningberg, and Christian Tschudin. A Virtualized Link Layer with Support for Indirection. In *FDNA 2004*.

[32] Albert Greenberg, Gisli Hjalmtysson, David A. Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. A clean slate 4d approach to network control and management. volume 35, pages 41–54, New York, NY, USA, 2005. ACM.

[33] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martin Casado, Nick McKeown, and Scott Shenker. Nox: Towards an operating system for networks. In *ACM Sigcomm Computer Communication Review*, 2008.

[34] Jonathan Hui. *An Extended Internet Architecture for Low-Power Wireless Networks–Design and Implementation*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 2008.

[35] Jonathan Hui and David Culler. Ip is dead, long live ip for wireless sensor networks. In *the 6th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2008.

[36] Xiafan Jiang, Stephen Dawson-Haggerty, Prabal Dutta, and David Culler. Design and implementation of a high-fidelity ac metering network. In *Proceedings of the 8th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN'09) Track on Sensor Platforms, Tools, and Design Methods*, 2009.

[37] Xiaofan Jiang, Prabal Dutta, David Culler, and Ion Stoica. Micro power meter for energy monitoring of wireless sensor networks at scale. *In submission*, 2006.

[38] Xiaofan Jiang, Joseph Polastre, and David Culler. Perpetual environmentally powered sensor networks. *IEEE SPOTS*, 2005.

[39] D. Johnson, Y. Hu, and D. Maltz. RFC 4728: The dynamic source routing protocol (dsr) for mobile ad hoc networks for ipv4, 2007.

[40] Dilip Joseph and Ion Stoica. Modeling middleboxes. In *IEEE Network Special Issue on Implications and Control of Middleboxes in the Internet*, 2008.

[41] Dilip Joseph, Arsalan Tavakoli, and Ion Stoica. A Policy-aware Switching Layer for Data Centers. Technical report, EECS Dept., University of California at Berkeley, June 2008.

[42] Dilip Joseph, Arsalan Tavakoli, and Ion Stoica. A policy-aware switching layer for datacenters. In *SIGCOMM*, 2008.

[43] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[44] P. Levis, A. Tavakoli, and S. Dawson-Haggerty. Overview of existing routing protocols for low power and lossy networks, 2008.

[45] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *SenSys '03: Proceedings of the 1st*

international conference on Embedded networked sensor systems, pages 126–137, New York, NY, USA, 2003. ACM.

[46] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, and David Culler. Tinyos: An operating system for wireless sensor networks. *Ambient Intelligence*, 2005.

[47] Philip Levis, Neil Patel, David Culler, and Scott Shenker. Trickle: A self-regulating algorithm for code maintenance and propagation in wireless sensor networks. In *First USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI)*, 2004.

[48] Philip Levis, Arsalan Tavakoli, and Stephen Dawson-Haggerty. Overview of existing routing protocols for low power and lossy networks. In *IETF Internet Draft*, 2009.

[49] Ajay Mahimkar, Jasraj Dange, Vitaly Shmatikov, Harrick Vin, and Yin Zhang. dFence: Transparent Network-based Denial of Service Mitigation. In *NSDI 2007*.

[50] Alan Mainwaring, Joseph Polastre, Robert Szewczyk, David Culler, and John Anderson. Wireless Sensor Networks for Habitat Monitoring. In *Proceedings of the ACM International Workshop on Wireless Sensor Networks and Applications*, September 2002.

[51] Yun Mao, Feng Wang, Lili Qiu, Simon Lam, and Jonathan Smith. S4: Small state and small stretch routing protocol implementation.

[52] Yun Mao, Feng Wang, Lili Qiu, Simon S. Lam, and Jonathan M. Smith. S4: Small state and small stretch routing protocol for large wireless sensor networks. In *NSDI*, 2007.

[53] J. Martocci, Pieter De Mil, W. Vermeylen, and Nicolas Riou. Building automation routing requirements in low power and lossy networks. In *IETF Internet Draft*, 2009.

[54] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turn. Openflow: Enabling innovation in campus networks. In *ACM Sigcomm Computer Communication Review*, 2008.

[55] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler. RFC 2922: Transmission of ipv6 packets over ieee 802.15.4 networks, 2007.

[56] Jorge Ortiz, Chris R. Baker, Daekyeong Moon, Rodrigo Fonseca, and Ion Stoica. Beacon location service: A location service for point-to-point routing in wireless sensor networks. In *International Conference in Information Processing in Sensor Networks: Special Track on Platform Tools and Design Methods for Network Embedded Sensors*, 2007.

[57] Vern Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23–24):2435–2463, 1999.

[58] Charles E. Perkins and Elizabeth M. Belding-Royer. Ad-hoc on-demand distance vector routing. In *WMCSA*, pages 90–100, 1999.

[59] K. Pister, P. Thubert, S. Dwars, and T. Phinney. Industrial routing requirements in low power and lossy networks. In *IETF Internet Draft*, 2009.

[60] K. S. J. Pister and L. Doherty. Tsmp: Time synchronized mesh protocol. In *Parallel and Distributed Computer and Systems*, 2008.

[61] Joseph Polastre, Robert Szewczyk, and David Culler. Telos: Enabling ultra-low power wireless research. *IEEE SPOTS*, 2005.

[62] E. Rosen, A. Viswanathan, and R. Callon. RFC 3031: Multiprotocol label switching architecture, 2001.

[63] Kannan Srinivasan, Prabal Dutta, Arsalan Tavakoli, and Philip Levis. Understanding the causes of packet delivery success and failure in dense wireless sensor networks.

Technical Report SING-06-00, Computer Science Department, Stanford University, 2006.

[64] Kannan Srinivasan, Maria Kazandijeva, Saatvik Agarwal, and Philip Levis. The β-factor: Measuring wireless link burstiness. In *Proceedings of the 6th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2008.

[65] Thanos Stathopoulos, Lewis Girod, John Heideman, Deborah Estrin, and Karen Weeks. Centralized routing for resource-constrained wireless sensor networks (sys 5), 2006.

[66] Ion Stoica, Daniel Adkins, Shelley Zhuang, Scott Shenker, and Sonesh Surana. Internet Indirection Infrastructure. In *SIGCOMM 2002*.

[67] Arsalan Tavakoli, Martin Casado, Teemu Koponen, and Scott Shenker. Applying nox to the datacenter. In *Proceedings of the 8th Workshop on Hot Topics in Networks (HotNets VIII)*, 2009.

[68] Gilman Tolle, Joseph Polastre, Robert Szewczyk, David Culler, Neil Turner, Kevin Tu, Stephen Burgess, Todd Dawson, Phil Buonadonna, David Gay, and Wei Hong. A macroscope in the redwoods. *ACM Sensys*, 2005.

[69] Michael Walfish, Jeremy Stribling, Maxwell Krohn, Hari Balakrishnan, Robert Morris, and Scott Shenker. Middleboxes No Longer Considered Harmful. In *OSDI 2004*.

[70] Geoff Werner-Allen, Konrad Lorincz, Jeff Johnson, Jonathan Lees, and Matt Welsh. Fidelity and yield in a volcano monitoring sensor network. *USENIX OSDI*, 2006.

[71] Alec Woo, Terrence Tong, and David Culler. Taming the underlying challenges of multihop routing in sensor networks. In *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems*, Los Angeles, CA, November 2003.

[72] Hong Yan, David A. Maltz, T. S. Eugene Ng, Heman Gogineni, Hui Zhang, and Zheng Cai. Tesseract: A 4d network control plane. In *In ACM/USENIX NSDI*, 2007.