A UNIFIED DATA-CENTRIC APPROACH TOWARDS AN EXTENSIBLE INTERNET ARCHITECTURE

Yun Mao

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania in Partial
Fulfillment of the Requirements for the Degree of Doctor of Philosophy

2008

Jonathan M. Smith and Boon Thau Loo
Supervisors of Dissertation

Rajeev Alur
Graduate Group Chairperson

Dedication

The Wenchuan earthquake occurred on 12 May 2008 in Sichuan province of China. It took away 69,196 lives as of July 6, and left millions of people homeless. I dedicate this dissertation to those people who passed away in this tragedy. May their souls rest in peace.

Acknowledgements

子在川上曰：“逝者如斯夫，不舍昼夜。”

---《论语》子罕篇 [1]

Looking back at the past six years during which I have been in the grad school at Penn, it is amazing to see how things have changed and how many people have accompanied, supported and influenced me. I could not have come through this journey without them.

First, I would like to thank my supervisors Jonathan M. Smith and Boon Thau Loo. Jonathan gave me enough room to be creative, but always kept me on track. He also gave me tremendous help on improving my writing, and, in particular, presentation skills. Boon inspired me with his passion, energy and talent on research. We had countless stimulating discussions that led to new ideas. They both helped to shape my research ideas and thesis work. I could not imagine to have any better advisor duo for me. You are the greatest!

I would also like to thank the following faculty members at Penn: Zack Ives enlightened me and showed me a broader perspective on my dissertation topic. He also impressed me on how an excellent writing skill can make a difference in presenting ideas. I also thank him for serving as my thesis committee chair and WPE-II committee chair; Steve Zdancewic co-authored my first paper with me since I came to Penn, and stimulated my interests in programming languages; Lawrence Saul introduced machine learning to me and surprised me with how much “five lines of MATLAB code” could do; Honghui Lu, as my first-year mentor and my friend, helped me on so many things to get started when I first came to Penn; Roch Guerin, Susan Davidson and Val Tannen spent a considerable amount of time to give me feedback on my job talk; Fernando Pereira, Milo Martin, Amir Roth, E Lewis, Matt Blaze, and Lyle Ungar provided many constructive suggestions on my

[1] English translation: standing by a river, Confucius said: “Time goes on and on like the flowing water in the river, never ceasing day or night!”

conference talks.

I spent one year visiting the University of Texas at Austin from 2005 to 2006. I would like to thank Lili Qiu for hosting me in her research group. She inspired me on many aspects in wireless networking. It was a lot of fun to work with her and her students. I am also grateful for Scott Nettles's generous support in providing me office space and official affiliation.

Internship is also part of my life during my graduate study. I thank Shaw Chuang at VMWare, and Hani Jamjoom, Shu Tao, and Nikos Anerousis at IBM Research for providing me such wonderful experiences.

I give my special thanks to my classmates, friends, and roommates Peng Li and John Blitzer. We came to Penn at the same year and went though so much joy and pain together. My thanks also go to the guys from the DSL labs, especially to Björn Knutsson and Dekai Li who helped me to move forward in my research during my first several years.

My research is funded in part by the following funding agencies and sources: National Science Foundation under contract CCR02-09024, CNS-0721845, DARPA under contract F30602-99-1-0512, BBN PO 9500009010, Air Force FA8750-07-C-0169, and the Olga and Alberico Pompa Chair Fund. I appreciate their support.

Last but not the least, I thank my family for their unlimited support and love. My parents Hanshu Mao and Youju Jin are always there for me. Dad: I assure you that it is *not* a coincidence that you taught me programming in BASIC 20 years ago with that bulky Epson “laptop” and now I obtain my Ph.D. degree in Computer Science. My wife Peng Bi, takes the best care of me for all these years. I was always her top priority especially when paper deadlines were close. Without her support I could not possibly come anywhere close to finishing the dissertation.

ABSTRACT

A UNIFIED DATA-CENTRIC APPROACH TOWARDS AN EXTENSIBLE INTERNET ARCHITECTURE

Yun Mao

Jonathan M. Smith and Boon Thau Loo

The Internet's role is changing dramatically, from a means of connecting together PCs and servers to a ubiquitous communications medium interconnecting mobile personal devices, environmental sensors, and Web services. As new services (voice, video, emergency response, *etc.*) are being deployed on the infrastructure, there have been increased demands on extending the existing Internet architecture for new capabilities, such as efficient and resilient routing among mobile and wired nodes, location of proximity-based services, and wide-area service discovery and composition.

This dissertation presents an extensible Internet architecture Mosaic— based on *declarative* languages and *composable* views over router, network and host state at different layers—to meet the demand of the emerging network services. The proposed architecture explicitly separates logical state representation and acquisition from physical implementation, to enable more extensible and adaptive protocols and distributed systems, because programmers can focus on high-level logical operations whose implementations may be separately and transparently optimized.

The dissertation makes the following contributions:

- A unified network architecture (Mosaic) under which new networks can be developed, deployed, selected, and dynamically composed according application and administrator needs.
- A declarative programming language (*Mozlog*) to concisely specify high-level network protocol specifications.
- A runtime system prototype that can translate *Mozlog* specifications into efficient implementations.

To validate the approach, we evaluate the work in the context of a heterogeneous, dynamic Internet environment where end hosts are connected via both wired and wireless media and have diverse application requirements. We demonstrate that in Mosaic, new network services can be readily introduced by either concisely specifying protocols in *Mozlog* or dynamically selecting and composing existing network services at low overheads.

Contents

Dedication **ii**

Acknowledgements **iii**

1 Introduction **1**

1.1 Motivation 1

1.2 Approach Overview 2

1.3 Evaluation of the Hypothesis 3

1.4 Summary of Contributions 4

1.5 Organization 5

2 Background **6**

2.1 Declarative Networking 6

2.2 Network Composition 7

2.2.1 Overlay Networks 8

2.2.2 Data plane composition 10

2.2.3 Control plane composition 11

2.2.4 Making Composition Decisions 12

2.2.5 Composition Challenges 13

2.3 Summary 14

3 MOSAIC: An Extensible Internet Architecture **15**

3.1 Overview 15

3.1.1 Infrastructure 15

3.1.2 MOSAIC Engine . . . 16
3.1.3 Directory Service . . . 17
3.2 Programming New Network Services . . . 18
3.3 Composing Network Services . . . 18
3.3.1 Composition Specifications . . . 19
3.3.2 Composition Compilation . . . 20
3.3.3 Dynamic compositions . . . 22
3.4 Summary . . . 23

4 The Mozlog Language **24**
4.1 Language Design . . . 25
4.1.1 Flexible Naming and Addressing . . . 26
4.1.2 Data and Control Plane Integration . . . 28
4.1.3 Modularity and Composability . . . 29
4.1.4 Special Predicates . . . 31
4.2 Implementation . . . 33
4.2.1 Dataflow . . . 33
4.2.2 Compilation of CViews . . . 34
4.2.3 Optimizations . . . 36
4.2.4 Special Predicates . . . 37
4.2.5 Legacy Support . . . 38
4.3 Summary . . . 39

5 Programming Network Services **40**
5.1 Resilient Overlay Network . . . 40
5.2 Internet Indirection Infrastructure . . . 41
5.3 Proxy Location . . . 42
5.3.1 DHARMA . . . 43
5.3.2 ROAM . . . 44
5.3.3 Flexible Proxy Selection . . . 45
5.4 Customizable Routing . . . 46
5.5 Service Discovery . . . 47

5.6 Evaluation . . . 48
5.6.1 LAN Experiments . . . 48
5.6.2 Emulab Experiments . . . 50
5.7 Summary . . . 52

6 Composing Network Services **53**
6.1 Compiling Compositions . . . 53
6.1.1 Compilation Steps . . . 54
6.1.2 Layering . . . 55
6.1.3 Bridging . . . 58
6.1.4 Feature Interaction . . . 58
6.2 Composition Examples . . . 60
6.2.1 VoIP between Alice and Bob . . . 60
6.2.2 Dynamic Composition of Chord over IP and RON . . . 61
6.2.3 Multicast for Pub/Sub Services . . . 62
6.3 Evaluation . . . 63
6.3.1 Wide-area Composition Evaluation . . . 63
6.3.2 Dynamic Overlay Composition . . . 65
6.4 Summary . . . 67

7 Related Work **68**
7.1 Extensible Protocols and Networks . . . 68
7.2 Network and Service Composition . . . 71

8 Conclusions **73**
8.1 Conclusions . . . 73
8.2 Open Questions and Future Directions . . . 74
8.2.1 Network Feature Interaction . . . 74
8.2.2 Composition . . . 75
8.2.3 Performance . . . 75
8.2.4 Programming Paradigm . . . 76

A Mozlog Grammar and Syntax **77**

B Composition Specification **81**

B.1 Composition Specification Example 81

B.2 Abstract Syntax of Composition Specification 82

List of Tables

2.1 A list of representative overlay networks and their features 8

5.1 Overhead comparison in LAN 49

5.2 Overhead comparison in LAN between native TCP, SOCKS proxy in Mosaic, and SOCKS proxy in Mosaic with optimized dataflow 50

6.1 netAddress table at Alice 60

6.2 netAddress table at Alice's gateway 61

List of Figures

2.1 Overlay composition by bridging. 10
2.2 Overlay composition by layering. 11

3.1 An illustration of the Mosaic infrastructure. 15
3.2 The Mosaic engine for network layer protocols. 17
3.3 Graph of *i*3 layered over RON, and private networks of Alice and Bob bridged with RON. 19

4.1 The ping module in *Mozlog* CView 29
4.2 The ping module in *NDlog* 30
4.3 System dataflow with dynamic location specifiers. 33
4.4 The result of ping CView translation from *Mozlog* to *NDlog* 35
4.5 The result of ping caller translation from *Mozlog* to *NDlog* 35
4.6 An illustration of tail recursive optimization on Chord lookup. 36
4.7 Chord lookup in CView 37

5.1 RON control plane *Mozlog* program 41
5.2 *i*3 control plane *Mozlog* program 42
5.3 Success rate vs sample size for proxy location using DHARMA and ROAM. 51

6.1 Dynamic composition of Chord over two different underlays (IP and RON). 61
6.2 A publish/subscribe composition example using multicast. 62
6.3 Throughput comparison between overlay composition in Mosaic vs direct IP connection during network failure. Network failures were injected 30 seconds after experiment start, and removed after 30 additional seconds. 64

6.4 Lookup accuracy comparison between Chord over IP and Chord over RON. 66

6.5 Chord lookup performance during dynamic underlay network switching from IP to RON. 66

A.1 The grammar: program declarations 78

A.2 The grammar: functor declarations 79

A.3 The grammar: expressions 80

B.1 Abstract syntax of composition specification 83

Chapter 1

Introduction

1.1 Motivation

The Internet's role is changing dramatically, from a means of connecting together PCs and servers to a ubiquitous communications medium interconnecting mobile personal devices, environmental sensors, and Web services. As new services (voice, video, emergency response, *etc.*) are being deployed on the infrastructure, there have been increased demands on extending the existing Internet architecture for new capabilities, such as efficient routing among mobile and wired nodes, location of proximity-based services, and wide-area service discovery and composition.

Overlay networks [69] that use the existing Internet to provide connectivity for new services are deployable [70]. They also enable innovation. However, despite deployment at global scale and emerging support for legacy applications [42], overlay networks now face several hurdles. First, they are often optimized for a specific application and may not be useful in all contexts. Second, overlay networks are generally targeted at and limited to niche *vertical* domains (*e.g.*, mobility [99, 60], security [44], reliability [7]). Third, the networks do not normally interoperate or share their functionality. For example, resiliency [7] and mobility [84] provided by one overlay cannot easily be leveraged by other overlay networks. Recent proposals for "clean slate" redesign of the Internet itself will exacerbate this problem, as more and more overlays are proposed and implemented.

Example 1.1.1 *Alice and Bob use private networks behind separate NATs, and wish to communicate regularly via VoIP or video conferencing, occasionally sharing data from internal web servers*

with trusted friends. As Alice and Bob travel regularly, and their IP addresses change, continued contact and communications should be seamless.

Alice and Bob can use a combination of *i*3 [84] for NAT traversal, ROAM [99] for mobility, RON [7] for reliability, and if DoS attack prevention is important, a secure overlay such as SOS [44] can be added. One may argue that a custom overlay such as Skype [82] may address some of the needs of Alice and Bob. However, a monolithic approach does not easily accommodate future application needs and changing network conditions. For example, RON may be excessive for a network with limited failures, and hence it may be desirable to remove it; whereas, in a partially-connected network, epidemic routing [90] would be desired. Further, Alice and Bob may require session-layer mobility support, hence requiring DHARMA [60] instead of ROAM [99].

Combining overlays to achieve desired capabilities is challenging in practice. One must first have a *composable* network architecture, unlike today's Internet. The architecture should provide interoperability among those separately designed and implemented overlay networks, and provide the mechanics of automatically interconnecting the overlays. Then, one must also identify combinations of overlays that can work together and provide the right set of capabilities. Emerging systems such as OCALA [42] have shown that *bridging* between different overlays requires significant "glue code." *Layering* one overlay over another is generally not even feasible yet, as each layer assumes it is running directly over IP.

1.2 Approach Overview

In this dissertation, we present a new point in the design space of network architectures that aims to achieve extensibility based on the application of database techniques to the networking domain. We propose a unified, extensible data-centric network architecture that allows for (1) extensible capabilities for *logical* naming of data in the form of *views*[1], (2) unified declarative queries for *distributed data management and state acquisition.* We argue that our approach enables (1) rapid authoring and deployment of new network services, either in the form of overlay networks on existing IP networks or a clean slate network redesign, (2) application-aware adaptivity to *select* and *compose* overlay networks to meet application needs, and (3) seamless support for legacy applications within the infrastructure.

[1] In database terminology, a view is a virtual or logical table composed of the result set of a query.

Logical-physical data model separation and distributed data acquisition and transformation are the cornerstones of the modern database field [37, 57, 33]. Hence, we propose that a *declarative*, database-inspired architecture using *query languages* is an ideal interface to these capabilities. Declarative languages provide both optimization and strong static verification possibilities, and declarative query languages focus on distributed data acquisition and transformation. Moreover, because such languages allow for very general definitions and are concise, they are much more extensible over time. Views can be composed, new queries and requests can be deployed, and new optimization or processing techniques can be invented and deployed without breaking compatibility. Such properties differentiate our approach from work on Active Networks [87], which typically used general programming models and only had a limited treatment of distributed coordination and data acquisition.

Our architecture uses *declarative networks* to build extensible network architectures that achieve a good balance of flexibility, performance and safety [56, 55]. Declarative networks are specified using distributed recursive query languages. Queries are executed using a distributed query processor to implement the network protocols, and continuously maintained as distributed views over existing networks and host state.

Declarative languages such as *NDlog* [55] are a natural and concise way to implement a variety of routing protocols and overlay networks. For example, traditional routing protocols can be expressed in a few lines of code [56], and the Chord [85] distributed hash table in 47 lines of code [55]. When compiled and executed, these declarative networks perform efficiently relative to imperative implementations. The orders of magnitude reduction in code size significantly increases programmer productivity. Moreover, these declarative specifications allow mobile services to be easily composed, and added to the infrastructure.

1.3 Evaluation of the Hypothesis

The key hypothesis explored in this dissertation is that a data-centric declarative architecture Mosaic is readily extensible to introduce new network services, yet at a low overhead when compared to a conventional network architecture. The remainder of the dissertation explores this hypothesis by: defining Mosaic; describing its specification language *Mozlog* and its implementation; demonstrating how it can be used to introduce new services by specifying them in *Mozlog*, and dynamically

selecting and composing existing services; and presenting evaluation results of the implementation.

To evaluate the hypothesis, several questions shall be answered in the dissertation:

- Is MOSAIC extensible enough to rapidly introduce emerging network services? In this dissertation, we build various network services, including mobility, reliability, multicast, and confidentiality, at different layers and demonstrate their feasibility in the context of a heterogeneous, dynamic Internet environment.

- Does the unified architecture achieve its goal of seamless application-aware selection and composition of network services? To assess its effectiveness, we evaluate it in the context of overlay networks, where services are realized and deployed. The runtime composition of existing overlay networks delivers the benefits that none of the existing ones can provide alone.

- What is the impact of the architecture on performance? Adopting the MOSAIC architecture might degrade network performance. We study the performance overhead via macro and micro-benchmarks in a controlled LAN environment, an emulated distributed environment in Emulab, and a realistic WAN environment in PlanetLab.

1.4 Summary of Contributions

The main finding of this research is that MOSAIC is able to introduce a diverse group of network services rapidly at a low cost using data-centric specification and composition. Specifically, the dissertation makes the following contributions:

- A data-centric network architecture (MOSAIC) under which new network services can be developed, deployed, selected, and dynamically composed according application and administrator needs.

- A declarative programming language (*Mozlog*) to concisely specify high-level network protocol specifications.

- A runtime system prototype that can translate *Mozlog* specifications into efficient implementations.

1.5 Organization

The remainder of this dissertation presents the Mosaic architecture, its programming language *Mozlog*, and its use cases and evaluation of its effectiveness.

Chapter 2 describes the background of the dissertation work. This includes the concept of declarative networking and the options for extending network services by composing existing services. Chapter 3 presents an architectural overview of the Mosaic architecture. Chapter 4 describes the *Mozlog* language, the compiler details and the runtime system implementation. Chapter 5 uses several concrete examples to demonstrate how to use *Mozlog* language to introduce new network services. Chapter 6 demonstrates services introduction by composing existing network services. Chapter 7 summarizes related work. Finally, Chapter 8 concludes the dissertation and discusses future directions.

Chapter 2

Background

Given that the Mosaic architecture is built upon declarative networking, we begin with a brief overview of its capabilities and its query language *NDlog* in this chapter. Then we introduce the concept of *network composition*, as an important method to extend network services in the Mosaic architecture.

2.1 Declarative Networking

The high level goal of *declarative networks* is to build extensible architectures that achieve a balance between flexibility, performance and safety. Declarative networks are specified using *Network Datalog* (*NDlog*), which is a distributed recursive query language used for querying network graphs[1].

Declarative queries are a natural and compact way to implement a variety of routing protocols and (overlay) networks. For example, traditional routing protocols such as path vector and distance-vector protocols can be expressed in a few lines of code [56], and the Chord distributed hash table in 47 lines of code [55]. When compiled and executed, these perform efficiently relative to imperative implementations.

NDlog is based on Datalog [73]: a Datalog program consists of a set of declarative *rules*. Each rule has the form:

```
p :- q1, q2, ..., qn.
```

[1] The authors of the declarative networking work have used both *NDlog* [54, 53] and *OverLog* [55, 15] as the language name, mostly interchangeably. In this dissertation, we consider them synonymous and use *NDlog* for consistency.

It can be read informally as "`q1` and `q2` and ... and `qn` implies `p`". Here, `p` is the *head* of the rule, and `q1`, `q2`,...,`qn` is a list of *literals* that constitutes the *body* of the rule. Literals are *predicates* with *attributes* , which are bound to variables or constants by the query, Boolean expressions that involve function symbols (including arithmetic) applied to attributes, or Assignment statements.Predicates in Datalog are typically relations, although in some cases they may represent functions. The predicates in traditional Datalog rules are relations, and we will refer to them interchangeably as predicates, relations, or tables.

Datalog rules can refer to one another in a cyclic fashion to express recursion. The order in which the rules are presented in a program is semantically immaterial; likewise, the order predicates appear in a rule is not semantically meaningful. Commas are interpreted as logical conjunctions (*AND*). The names of predicates, function symbols, and constants begin with a lowercase letter, while variable names begin with an uppercase letter.

Network Datalog (*NDlog*) is a distributed variant of traditional Datalog, primarily designed for expressing distributed recursive computations common in network protocols. We illustrate *NDlog* using a simple example of two rules that compute all pairs of reachable nodes:

```
r1 reachable(@S,D) :- link(@S,D).
r2 reachable(@S,D) :- link(@S,Z), reachable(@Z,D).
```

The rules `r1` and `r2` specify a distributed transitive closure computation, where rule `r1` computes all pairs of nodes reachable within a single hop from all input links, and rule `r2` expresses that "if there is a link from `S` to `Z`, and `Z` can reach `D`, then `S` can reach `D`." By modifying this simple example, we can construct more complex routing protocols, such as the distance vector and path vector routing protocols.

NDlog supports a *location specifier* in each predicate, expressed with `@` symbol followed by an attribute. This attribute is used to denote the source location of each corresponding tuple. For example, all `reachable` and `link` tuples are stored based on the `@S` address field. The output of interest is the set of all `reachable(@S,D)` tuples, representing reachable pairs of nodes from `S` to `D`.

2.2 Network Composition

Network composition is the act of combining distinct parts or elements of existing networks to create a new network with new functionalities. Overlay composition is network composition of

Feature	Representative overlays
resiliency	RON, Detour, SOSR
mobility	*i*3, DHARMA, Warp
scalable lookup	Chord, Pastry, CAN, Tapestry, OpenDHT
multicast	Overcast, ESM, PPLive
anycast	Akamai, CoralCDN, CoDeeN, GIA
security	SOS, OverDoSe, Cashmere

Table 2.1: A list of representative overlay networks and their features

overlay networks, and so results in a new overlay network. In this section, we introduce the concept and background of *network composition* in overlay networks, as an important method to extend network services in the Mosaic architecture, then we consider composition of overlays along both data plane and control plane.

2.2.1 Overlay Networks

An overlay network is a virtual network of nodes and logical links that is built on top of an existing network. It aims to implement and provide a network service that is not available in the existing network. Most of the logical networks are overlay networks in the sense that they are over-laid on top of other physical networks. In a broad sense, the most widely used and successful overlay network is the Internet. In this dissertation, we use the term *overlay networks* to refer to the logical networks that are layered on top of IP.

Different overlay networks provide different features or functionalities. Table 2.1 and the following list provides a sampling of overlay networks proposed in the previous decade, categorized by their features:

- Resiliency: the Internet is sometimes subject to transient failures, *e.g.* due to BGP misconfiguration, fiber link cut, etc. By exploiting the redundancy in the Internet path, many projects propose to use overlay routing to quickly react to routing failures and provide a resilient network service. Such systems include, but not limited to Resilient Overlay Network (RON) [7], Detour [80], Scalable One-hop Source Routing (SOSR) [32], etc.

- Mobility: the Internet was not designed with mobile devices in mind. When a mobile device, such as a PDA or a smartphone, wants access to the Internet at different network attachment points, its IP address changes, which is disruptive to the communication. While Mobile IP [39, 68] tries to address this problem within the IP protocol, there are many proposals that

try to tackle the problem as an overlay network, *e.g.* ROAM [99] based on Internet Indirection Infrastructure (*i*3) [84], Distributed Home Agent for Robust Mobile Access (DHARMA) [60], and Warp [96], etc.

- Scalable lookup: the lookup service is also known as a Distributed Hash Table (DHT). The goal is to locate the node that holds responsible for the given key *consistently* and in a scalable fashion in an overlay network with a large number of nodes. Such proposals include Chord [85], Pastry [78], CAN [74], Tapestry [97], and OpenDHT [75].

- Multicast: Multicast is frequently used in distributing multimedia contents, such as video-on-demand or live broadcast systems. IP multicast capability was built into the IP protocol suite. Unfortunately, it is not widely deployed and is generally not available as a service for the average end users, due to security and deployment complexity concerns [71]. Solutions based on overlays such as Overcast [41], ESM [35] and PPLive [36] tend to address these issues.

- Anycast: anycast is a network addressing and routing scheme whereby data is routed to the "best" destination as viewed by the routing topology. It is mainly used to implement DNS or content distribution systems, such as Akamai [4], CoDeeN [91], and CoralCDN [26, 25], etc.

- Security: a number of overlays provide services with security-related features. Perhaps one of the simplest examples is a secure VPN service, where network traffic is encrypted by SSL or IPSec to provide confidentiality. Alternatively, some overlays aim to prevent denial of service (DOS) attacks (such as Secure Overlay Service SOS [44] and OverDoSe [81]) or to provide anonymity, such as Cashmere [98].

The list above is by no means a complete or the only taxonomy of features provided by all overlay networks. However, it does demonstrate that the features that can be provided by overlay services are rich. Besides, they usually conform to a similar interface with `send/receive` APIs. However, the semantics may be completely different. For example, the addresses are logical rather than physical in many cases: in mobility, addresses are not bind to physical networks; in anycast and multicast, a group of nodes may be grouped into a single address; DHTs, usually understood as having a `get/put` API, may also be approximate to routing towards a the logical keys as the addresses.

Unfortunately, those overlays are designed for application specific domains. If a user demands two or more features *simultaneously* for the same application, most likely none of the existing proposed overlay solutions satisfy the requirements. A desirable approach is to compose existing overlays together to provide a combination of the wanted features. We consider composition of overlays along both data plane and control plane.

2.2.2 Data plane composition

The data planes of two overlay networks can be composed horizontally by *bridging* between the networks, or they can be composed vertically by *layering* one overlay over the other.

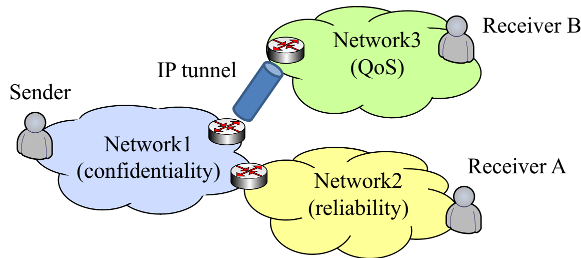

Figure 2.1: Overlay composition by bridging.

In *bridging* (see Figure 2.1), each overlay network runs on top of the same substrate (*e.g.*, the IP network) directly. However, for a variety of reasons (*e.g.*, sending from a wireless to a wired network), it may be necessary to send a packet across multiple overlay networks to reach the receiver. This is usually done via a *gateway* node that belongs to both networks. If such gateways do not exist, two nodes from each network need to be connected via an IP tunnel to route packets. In Figure 2.1, a sending laptop using wireless may use an overlay that provides confidentiality to route traffic over the wireless links, then use an overlay with reliability guarantees to deliver important but not time-sensitive data to receiver A, while using a QoS overlay to deliver multimedia traffic to receiver B.

In *layering*, logically a packet is routed within a single data plane of an existing overlay network. However, the data paths between the nodes inside the overlay may be constructed on top of other overlay networks, rather than the Internet. For example, RON [7] only works for nodes that have publicly routable IP addresses. As shown in Figure 2.2, by composing RON on top of another overlay protocol that enables NAT (Network Address Translation) traversal, such as *i*3, nodes

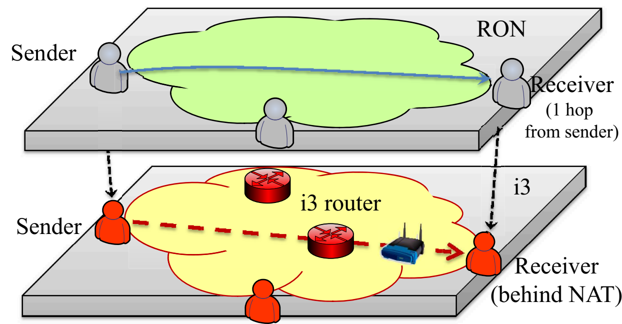

Figure 2.2: Overlay composition by layering.

behind NAT should be able to join the RON network.

We note that the two data plane compositions listed above are not mutually exclusive; some data composition scenarios can combine both layering and bridging.

2.2.3 Control plane composition

An overlay network's control plane can be layered over either the data plane or the control plane of another overlay. For example, it is possible to build the control message channels of DHT protocols such as Chord over the data plane of RON. Typically, the failure detection components of DHTs assume that hosts unreachable via IP are dead. In fact, some hosts may be alive and functioning, but temporary network routing failures may create the illusion of node failure to part of the overlay nodes. If the network failure happens intermittently, churn rate is increased and may create unnecessary state inconsistency. Using a resilient overlay like RON can overcome some of the network failures to reduce churn.

Some overlay network protocols have complex, layered control planes. For example, both *i*3 [84] and DOA [11] (a delegation-oriented architecture that facilitates deployments of middleboxes) use DHTs for either forwarding or lookup. RON [7] (an overlay providing resilient packet delivery) and OverQoS [86] (an overlay providing QoS guarantee) heavily depend on measurements of underlying network performance characteristics such as latency and bandwidth. When overlay networks are built from scratch over IP, it is conceivable that different logical overlays built on the same physical IP topology may duplicate the effort to maintain DHTs or perform network measurements. Nakao, *et al.* [65], observed that on PlanetLab, each node had 1GB outgoing ping traffic daily: many overlay

networks running on the same node were probing the same set of hosts without coordination. Such duplicated probing traffic can be wasteful, and interactions between probe traffic may introduce measurement error. A composition-driven approach is to build smaller elements that provide well defined interfaces (*e.g.*, OpenDHT [75] for DHT lookup and iPlane [58] for measurement) so that they can be easily composed with upper layer overlay network control planes to share rather than compete for resources.

2.2.4 Making Composition Decisions

With the proliferation of overlay networks and different possibilities for composition amongst them, a natural question to ask is where the decision should be made of which overlays to compose, and how should they be composed.

We argue that it should be a collaborative effort between overlay providers and end users. Overlay providers should decide the control plane composition at design time. In addition, providers should provide service descriptions as meta-data for the overlay networks they have deployed, which articulate the functionalities and composition requirements. For example, if an overlay network provides confidentiality by encrypting all the traffic, it makes little sense to compose it on top of another overlay that does compression to save bandwidth consumption, because the compression needs redundancy in the data, which do not exist in well encrypted packets. Such meta-data is useful in helping end users make the right composition decisions.

Users at the end-points should take control of the data path composition, given existing functionalities of the overlay networks they have access to. In particular, a sender should be able to choose which overlay networks to traverse before her traffic reaches the receiver(s) and a receiver should be able to dictate which overlay networks the packets come from. We argue that the control decisions made by the end points agrees with the end-to-end argument [79]. End points have better knowledge of applications, so they can be expected to intelligently compose overlay network resources to achieve application-specific design goals. This contrasts with a single monolithic overlay network attempting to optimize all aspects of the application requirements, such as efficiency, reliability, mobility, security, etc.

To be clear, we do not require that users must be involved in every composition, but rather that the end-point should be the locus of control. Ideally, software should automatically discover available overlay networks and make intelligent composition decisions on behalf of users, perhaps

with their guidance.

2.2.5 Composition Challenges

From an architectural point of view, there are several challenges to building an extensible architecture to support overlay composition:

- **Interoperability**: Because the data plane composition decision is made by end users, it happens dynamically, at runtime. An overlay protocol cannot assume that the underlay is IP. Instead, it should be able to inter-operate with other overlays, which may not even exist at the time it is deployed. Unfortunately, because most overlay network designers have IP in mind as the substrate, the implementation is tightly coupled with the IP protocol itself. A simple example is that in some implementation, an IP address is stored in a 32-bit integer. If we were to run the same protocol on a different network, even for protocols like IPv6 without much architectural difference, a new implementation is needed, which leads to the entire deployment process of testing, router upgrading, user adoption, etc.

- **Feature interaction**: A rigorous composition specification method requires a proof rule asserting that if each component behaves correctly in isolation, then it behaves correctly in concert with other components. Such a rule is subtle because a component need behave correctly only when its environment does, and each component is part of the others' environments [1]. In network composition, different network protocols provide different services, with different approaches. Two networks are *incompatible* if they provide conflicting goals or use conflicting approaches. To compose incompatible networks together may negate the effectiveness of the individual services. For example, if network A provides a routing service with minimal latency, while network B provides a routing service with maximum bandwidth, layering them together as a composed network may deliver neither feature. Consider another example where two end-to-end services use encryption and compression respectively. If a packet is encrypted before it is compressed, the compression algorithm can hardly achieve any benefit, while doing compression before encryption is perfectly fine. That is, the order of the service composition may also lead to undesirable feature interaction. A composable network architecture needs to provide the opportunity to validate the compatibility of a proposed network composition and guide the users towards effective composition.

2.3 Summary

In this chapter, we introduced the background of declarative networking and the *NDlog* programming language, upon which our MOSAIC network architecture is built on. We then described the concept of *network composition* as an important methodology to extend network services. We classified it into *data plane composition* and *control plane composition* and gave motivating examples respectively. The challenges of network composition is also discussed. In the following chapter, we present the MOSAIC architecture to address some of the challenges.

Chapter 3

MOSAIC: An Extensible Internet Architecture

MOSAIC is an architecture to design, implement and deploy *composable* networks based on a data-centric declarative networking approach using a unified, data-centric declarative programming language. In this chapter, we provide an overview of the proposed MOSAIC architecture.

3.1 Overview

3.1.1 Infrastructure

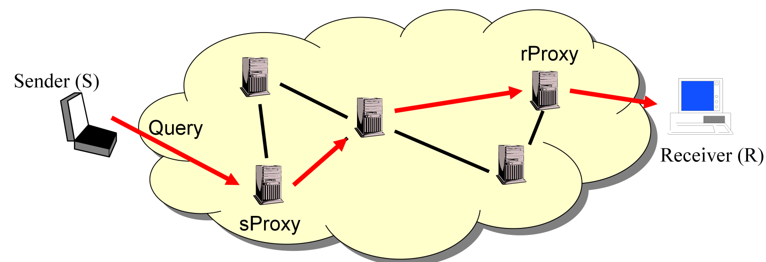

Figure 3.1: An illustration of the MOSAIC infrastructure.

The MOSAIC infrastructure consists of a network of nodes, which can either be integrated with existing communication infrastructures or run as an independent overlay network. The key distinction of MOSAIC from the existing Internet infrastructure is that the nodes or routers in the

network are no longer "dumb" devices, but enhanced with database query processing capabilities. Therefore, the infrastructure provides distributed query capabilities over all network routing state, device session state and system monitoring information—in the form of database tables as opposed to more traditional data structures. At the core of the MOSAIC infrastructure is a *declarative networking* engine [54, 56, 55, 53]. Declarative networking leverages a database query language for specifying and implementing network protocols, and employs a dataflow framework at runtime for communication and the maintenance of network state. The key idea is that declarative recursive queries [2, 73], which are used in the database community for querying graph structures, are a natural fit for expressing the properties of various network protocols. The primary goal of declarative networking is to greatly simplify the process of specifying, implementing, deploying and evolving a network design. In addition, declarative networking serves as an important step towards an extensible, evolvable network architecture that can support *flexible*, *secure* and *efficient* deployment of new network protocols.

On this infrastructure, several network services deployed in the form of overlays may co-exist, and are not necessarily deployed on all nodes. Individual overlay protocols are specified using the *Mozlog* declarative networking language, while compiled and executed in MOSAIC. Composed overlay networks will be instantiated by leveraging existing deployed overlays, either by layering (above or below) or bridging with them. In addition, private networks outside of the infrastructure will be bridged via public gateways with overlays deployed on this infrastructure.

At the edges of the infrastructure, there are *proxy nodes* that communicate directly with wireless or other embedded devices with limited processing capabilities, route messages on their behalf and provide location-based services. In Figure 3.1, the client issues a query that is used to identify the proxy node at the sender (sProxy) and receiver (rProxy), and a route is established based on the session requirements, device characteristics and infrastructure resource availability.

3.1.2 MOSAIC Engine

Figure 3.2 illustrates the MOSAIC engine from the perspective of a single node. MOSAIC is positioned at the network layer in the network stack to replace IP. It exposes a simple interface to the transport layer by providing two primitives: `send(DestAddress, Packet)` and `recv(Packet)`. In IP, a packet consists of an IP header with fixed format and a raw byte data as the payload. In MOSAIC, `Packet` is represented abstractly as a structured data element, which might be a set of scalar values or even

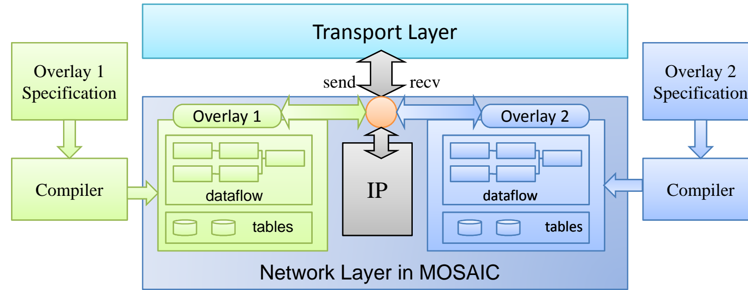

Figure 3.2: The MOSAIC engine for network layer protocols.

nested tuples. The encoding of this packet is up to the specific overlay protocol, and declarative mappings or transformations can convert between the packet formats of different overlays (see Chapter 4). `DestAddress` is a specially typed tuple, with the first attribute being the identifier of the overlay network to which the packet belongs. This identifier is used to demultiplex the send requests to different overlays or IP at the network layer. A `send` request will trigger a `recv` event at the node or nodes who own the `DestAddress` if the network successfully routes the packet.

3.1.3 Directory Service

For each overlay running on the infrastructure, there is a directory service that maintains the following information: (1) A *unique identifier* for the overlay; (2) The list of *physical nodes* that is currently executing the overlay; (3) The list of users who can utilize the overlay, and their privileges (*e.g.*, whether they can bridge with this overlay). These privileges are set by an overlay's owner; and (4) Additional meta-data that describes the overlay, such as its attributes, node constraints, etc. As part of the process of creating a composed overlay, the user may issue queries to the directory, searching for existing overlays that meet their criteria for composition.

The directory service may be provided either by a centralized server or in a distributed fashion [19, 10] for scalability. The design choice of the directory service is orthogonal to the MOSAIC architecture. In this dissertation, we focus on the use of a centralized server. We note that a centralized service is sufficient for maintaining the meta-data information for thousands of infrastructure nodes, as demonstrated by PlanetLab central [70].

There are two ways to create new network services in MOSAIC: (1) by specifying queries as network protocols to create new networks, and (2) by composing existing networks to create a

new network with a combination of existing functionalities. We discuss the two approaches in the following sections.

3.2 Programming New Network Services

In Mosaic, network protocol specifications are written in *Mozlog*, which is a data-centric declarative query language based on *NDlog* [55]. Mosaic takes a *Mozlog* program, compiles it into distributed dataflows [55], and deploys it to all nodes that participate the new network in the form of overlays. After it is deployed, the meta-data of the overlay information is stored at the directory service.

On each single node, multiple overlay networks may be hosted at the same time. The distributed dataflows compiled from the *Mozlog* program resemble the execution model of the Click modular router [47], which consists of elements that are connected together to implement a variety of network and flow control components. In addition, the elements include database operators (such as joins, aggregation, selections, and projections) that are directly generated from queries. Each local dataflow participates in a global, *distributed* dataflow across the network, with messages flowing among elements at different nodes, resulting in updates to local tables. The local tables store the state of intermediate and computed query results, including structures such as routing tables, the state of various network protocols, and data related to their resulting compositions. The distributed dataflows implement the operations of various network protocols. The flow of messages entering and leaving the dataflow constitute the network packets generated during query execution.

We describe the *Mozlog* language in detail in Chapter 4, and provide several case studies in Chapter 5.

3.3 Composing Network Services

To create a composed overlay network, a Mosaic user (*e.g.* a network administrator) first uses the directory service to locate overlay networks that meet their criteria for composition, and retrieves relative meta-data information. Second, the administrator creates a *composition specification*, which is a high-level graph-based description of the desired component overlay networks and their interactions. Then, the specification is compiled into the *Mozlog* language used by Mosaic's compiler, described in Section 6.1. As part of this process, new code is created that "glues" the compositions

together. Finally the generated *Mozlog* code is deployed to the physical nodes to start the new network, and the directory information is updated regarding the newly composed network.

3.3.1 Composition Specifications

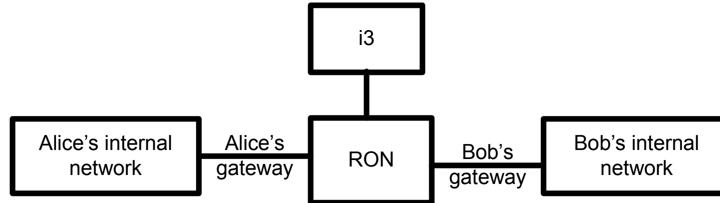

Figure 3.3: Graph of *i*3 layered over RON, and private networks of Alice and Bob bridged with RON.

Figure 3.3 shows a graphical representation of a composition specification, based on the example scenario introduced in Section 1.1. We chose XML as the internal representation to describe the graph (See Appendix B.1). Each module (node) represents a component overlay network (*e.g.*, *i*3 and RON) deployed on the infrastructure, or a private network. The links represent *connectors*, where vertical and horizontal links denote layering and bridging, respectively. Here, the *i*3 overlay is layered over RON; Alice and Bob's private networks are bridged to RON. In addition to a unique overlay identifier, each module configuration consists of the following:

- **Physical node constraints**: When the overlay is first deployed, the user who created the overlay can constrain the set of nodes on which the overlay may execute. This can be in the form of a prefix to indicate that nodes must be deployed on particular subnets, or enforce the inclusion of particular nodes (*e.g.* Alice's and Bob's gateways) must be on both the *i*3 and RON networks.

- **Attributes:** Each overlay network has properties that characterize its capabilities, including mobility, secure routing, NAT traversal, resilient routing, anonymity, private networks, etc. These properties can be queried by users to identify overlays that meet their requirements.

- **Code:** If a module is loaded for the first time, code can be included in the configuration. This can either be legacy code, or *Mozlog* specifications for declarative networks.

- **Default gateway:** Each module can specify a default gateway for bridging. In the absence of a specified gateway, the a common physical node sitting on both networks is selected to serve as the gateway.

- **Access control:** MOSAIC supports restrictions on which users can utilize an overlay, and their privileges (*e.g.*, layering above or below, and bridging, etc).

The connectors between modules have properties associated with them. Bridging (horizontal lines) must specify whether there are default gateways to be used, and whether tunneling is permitted. If two modules are specified to be bridged via a default gateway node, both overlays must run on the specified gateway. Layering (vertical lines) also has constraints on whether the overlay has to be layered on all or subset of the nodes. In this example, to get the full benefits of RON, all *i*3 nodes should utilize RON as their underlay. However, this is not strictly required: *i*3 nodes that do not run RON will default to using IP. For both bridging and layering, one can further specify whether some connections replace existing ones.

3.3.2 Composition Compilation

Once the composition is specified, a *composition compiler* is used to generate the *Mozlog* code that "glues" together different overlay networks based on the specifications. The compiler is a software deployed either at client-side, or as a service in conjunction with the directory service.

The compilation process can be performed in two different ways. First, a composition can *create* overlays, either from scratch where each module contains the code implementing each overlay, or incrementally where the new overlay is built on existing ones, *e.g.*, by adding new overlays over them, or bridging overlays via identified gateways. Creating overlays incrementally requires the composition specifications to refer to existing overlays by their unique identifiers. Second, a composition can also *modify* overlays, which involves replacing existing modules with new ones, and this requires connectors to indicate that they are replacing existing composition links.

Given the above mechanisms, we outline how layering and bridging can be achieved by compiling modules and connectors, and provide a detailed process description and examples in Section 6.1. The first step is to perform basic checks to ensure all the links are legal, based on the attribute constraints and physical node constraints. For example, one cannot layer one overlay over another if they are configured for completely disjoint sets of nodes. Two overlays cannot be bridged if their

bridge connector does not permit tunneling and the two overlays do not share any common node. Once validated, *Mozlog* rules for composition and all required overlay code are uploaded to relevant nodes for execution.

Layering

Layering of a control or data plane over another overlay's data plane is achieved by ensuring that every protocol uses *logical addresses* — rather than being bound to physical addresses. At runtime Mosaic will bind (or rebind) the upper layer's logical address to the underlay address. These bindings are stored in a separate table that can be updated to facilitate dynamic changes to layering.

Mosaic allows the control plane of one overlay network to layer over another overlay's control plane, accessing its internal state. Here, each overlay exports the state of its composable components, in the form of database logical views (query results presented as a named table). An example of such state is a distributed hash table's contents, which can be modeled as a relation with tuples associating keys and values. Importantly, accessing a neighboring protocol's state can be done within the overlays' specification language — there is no "impedance mismatch" between languages, and interoperability issues are minimal.

Bridging

Depending on requirements, bridging can be done either *pre-configured* or *on-demand* in Mosaic.

Pre-configured method. When the composition specification involves bridging multiple overlays, forwarding state is created on designated gateways based on the bridge connectors indicated in the composition specifications. When a sender sends a packet whose destination contains an address of an overlay in which the sender does not participate, Mosaic routes the packet to the gateway, which then continues to forward the packet along the bridged overlay. In addition to a static gateway, the sender can also use a pre-configured anycast service [43, 25] to select and route packets to one of the overlay nodes, preferably close in terms of network distance to the sender.

On-demand method. The sender utilizes source routing to explicitly describe the data path to the destination via designated gateways among different overlays found in the specification.

Alternatively, the gateway holds address translation state that uniquely identifies the flow between the sender and the receivers, it performs indirection. The on-demand mechanism enables user-driven dynamic bridging. We will describe several examples of such compositions in Section 6.2 using the *Mozlog* language.

3.3.3 Dynamic compositions

MOSAIC exploits *Mozlog*'s declarative model to facilitate *dynamic overlay composition*: since network definitions in MOSAIC separate specification from implementation, the system can (assuming the right constraints are met) freely replace either the IP or an existing overlay underneath one overlay network with a second overlay network—*i.e.*, it can *layer networks*. For example, the protocol used in RON is a modified link-state protocol, which is general enough to operate on any connected graph. The original RON implementation assumes IPv4 as a substrate, and hence it is hard-coded to use publicly routable IP addresses. In MOSAIC, protocols are written with a network-agnostic addressing scheme, so a RON overlay can instead use addresses from one or more lower-level overlay networks, provided they are reachable from one another. This allows MOSAIC to *dynamically switch* an existing overlay's underlay based on the network conditions, *e.g.*, an executing overlay that utilizes IP can dynamically layer itself over RON when routing losses are high, or further switch to an epidemic forwarding strategy when the network is disconnected.

Dynamic overlay switching in MOSAIC is achieved by changing the binding between an upper overlay's logical addresses and the underlying network and its (lower-level) addresses. This technique is overlay-agnostic. However, we must be careful to preserve application and overlay semantics. In particular, if dynamically switching maintains the *same endpoints* on route requests (as RON, above, does), then the switch is permissible. Likewise, if the lower overlay *state is not visible to the layers above*, and all endpoints provide the same functionality (*e.g.*, in a content distribution network), then the switch is also permissible. In other cases, we would need to re-architect the overlays and possibly the application to redistribute state over the new underlay, and to be tolerant of transient states.

3.4 Summary

In this chapter, we provided an overview of the proposed Mosaic architecture using the data-centric declarative approach. We described the infrastructure, where each node runs a Mosaic engine that executes the compiled *Mozlog* programs. Then we introduced two methods to extend network services in Mosaic: creating new network services in *Mozlog* programming and composing services over existing network services using high-level composition specification. In the following two chapters, we give concrete examples to demonstrate how to use these two methods respectively for rapid network service introduction.

Chapter 4

The Mozlog Language

Mozlog is the domain-specific programming language we design for the MOSAIC architecture. Protocols are written in *Mozlog* instead of traditional general purpose programming language like C or Java. Before introducing the language, we highlight the following design goals for the MOSAIC architecture:

- Interoperability: MOSAIC should provide interoperability among multiple overlay networks, including both the control planes and the data planes;
- Dynamic composition: the composition of overlay protocols should be able to changed on the fly by the application requirements;
- Reusability: overlays previously deployed should be able to compose with emerging overlays without change;
- Independence: MOSAIC should make as few assumptions as possible about the underlying networks, and should not be confined to network layer protocols;
- Conciseness: given above requirements, the language should remain concise to let the developers focus on the high-level protocol specification;
- Legacy support: legacy applications should not break despite the use of disruptive overlay networks.

In rest of this chapter, we provide the design and implementation of the *Mozlog* language and show how it helps MOSAIC to achieve those goals. The detailed language grammar and syntax can be found in Appendix A.

4.1 Language Design

Mozlog is derived from Datalog and *NDlog* which we have introduced in Chapter 2. In this section, we focus *Mozlog*'s distinct features that is required for the MOSAIC architecture. We categorize these features as follows:

- **Flexible naming and addressing:** *Mozlog* supports *dynamic location specifiers* whose types (*e.g.*, IP address or logical overlay identifier) are determined at runtime. In addition, location specifiers are decoupled from data attributes and made optional for local data. These two language extensions not only enable interoperability among multiple overlays, but provide multi-homing and mobility features (Section 4.1.1).

- **Data and control plane integration:** *Mozlog* provides language support for forwarding on the data plane. This provides extensibility at both the control and data plane, and hence provides flexible composition of different overlay features on either plane (Section 4.1.2).

- **Modularity and reusability:** *Mozlog* allows multiple declarative rules to be composed and modularized as *Composable Virtual Views* (*CViews*). This enables features of different overlays to be modularized, hence facilitating composition of different features, and improved resource sharing. As an additional benefit, *Mozlog* provides more concise specifications and better abstractions for timeouts and exception handling (Section 4.1.3).

- **Special predicates:** *Mozlog* provides several predicates for accessing the tun device and using TCP. This enables legacy application support at both the network and transport layers and creates the opportunity to build transport layer overlays (Section 4.1.4).

This section focuses purely on the *Mozlog* language. We discuss implementation details in Section 4.2. Detailed use cases and experimental analysis are provided in Chapter 5 and Chapter 6.

4.1.1 Flexible Naming and Addressing

Location specifiers in *NDlog* currently have two limitations. First, they are assumed to be IP addresses, hence limiting their usage to IP-based networks. As a result, it is not possible to express data placement in terms of overlay identifiers or differentiate data that belongs to different overlays. Second, location specifiers tightly couple data's attributes to its location, limiting each host to store only data at a unique network address. This prevents multi-homing, an important requirement when each physical host may be simultaneously associated with several logical overlay networks. Third, mobility is not supported since any change in IP address will invalidate all local tables.

To address these limitations, we make two modifications. First, we *decouple* each datum from its location specifier, and make the location specifier optional. Second, we associate all location specifiers with a runtime type.

Decoupling Location from Data

Mozlog predicates have the following syntax:

```
predicate@LocSpec(Attr1,  Attr2, ...)
```

For backward compatibility to *NDlog* and conciseness, two forms of syntactic sugar are provided too, where `predicate(Attr1,..., @AttrI, ..)` is equivalent to `predicate@AttrI(Attr1,..., AttrI, ..)` and `predicate(Attr1,Attr2, ..)` is equivalent to `predicate@LocalID(LocalID, Attr1,Attr2, ..)`. That is, in the absence of any location specifier, `predicate` is assumed to refer to local tuple.

For example, in the following rule,

```
a1 alarm@R(L, N) :- periodic(10),
    cpuLoad(L),
    nodeName(N),
    monitorServer(R),
    L>20.
```

`periodic` is a built-in local event that will be triggered every 10 seconds. The predicates `cpuLoad`, `nodeName`, and `monitorServer` are local tables. The rule specifies that for every 10 seconds, if the

CPU load is above the threshold 20, an `alarm` event containing the current load `L` and hostname `N` will be sent to the monitoring server `R`.

Decoupling data from its location enhances interoperability and reusability. Now multiple overlays can interoperate (*i.e.*, exchange state) by sending network-independent data tuples in a common data representation. Moreover, since these rules are rewritten in a location-independent fashion, they can be reused on different network types (*e.g.*, *i*3, RON, or IP). Finally, since it does not bind addresses to data, the language is friendly to mobility, where host movement (and hence resulting change in its IP address) does not invalidate its local tables.

Runtime Types for Location Specifiers

Our second modification involves adding support for runtime types to location specifiers. This feature is necessary for dynamically composing multiple overlays at runtime. Location specifiers are denoted by an `[oID::]nID` element, where `oID` is an optional unique string identifier for an overlay network, and `nID` is a mandatory overlay node identifier. For example, `i3::0x123456789I` denotes an *i*3 node with identifier `0x123456789I`, and `ron::"158.130.7.3:10000"` denotes a RON node with IP address 158.130.7.3:10000. In the absence of any overlay identifier, IP is assumed.

At runtime, Mosaic examines the location specifier of each tuple and routes it along the appropriate network. For example, a tuple with an IP address as a location specifier is sent directly via IP. On the other hand, a tuple designated for an overlay network is sent to the corresponding overlay network. To illustrate the flexibility of our addressing scheme, consider the CPU load monitoring example from Section 4.1.1. Rule `a1` can be rewritten as `a2`, in which the monitoring server `R` refers to an *i*3 key generated as a hash of its name `N` instead of an IP address:

```
a2 alarm@R(L, N) :- periodic(10),
    cpuLoad(L),
    nodeName(N),
    serverName(SN),
    L>20,
    Key := f_sha1(SN),
    R:=i3::Key.
```

Dynamic location specifiers enable bridging of different overlays easily. For example, a gateway

node `G` can physically host two overlay network nodes (one for *i*3 and another for RON), and is addressable via either network. A source routing specifier is used to perform forwarding via the gateway node. For instance, node `Dest` in RON with address `sr::[i3::Gateway_key, ron::Dest]` is reachable from all hosts in the *i*3 network. As an additional benefit, dynamic location specifiers enable addresses to be updated at runtime to switch between IP networks and various overlays. We provide a detailed example in Chapter 6.

4.1.2 Data and Control Plane Integration

Declarative networking previously focused on the control plane of networks. Overlay composition requires the integration of the data and control planes of multiple overlays. To achieve this, *Mozlog* enables declarative specification of the data plane behavior. Each overlay network has `send` and `recv` predicates that are used to specify data forwarding within an overlay. We provide an example based on the data plane of RON:

```
snd ron.send@Next(Dest, Packet) :- ron.send(Dest, Packet),
    localAddr(Local),
    Local!=Dest,
    ron.RT(Dest, Next).
rcv ron.recv(Packet) :- ron.send(Dest, Packet),
    localAddr(Local),
    Local==Dest.
```

Rule `snd` expresses that for all non-local `Dest` addresses, the data packet (`Packet`) is sent along the next hop (`Next`) which is determined via a join with RON's routing table (`ron.RT`) using `Dest` as the join key. These packets are then received via the rule `rcv` at node (`Dest`), which generates a `ron.recv(Packet)` event at `Dest`.

In *Mozlog*, the `send` and `recv` predicates are usually not directly used by other rules, but rather automatically invoked by the MOSAIC runtime engine when the location specifier type of a tuple matches the overlay. As a result, one can bridge the data planes of different overlays together, or layer the control plane of one overlay network over the data plane of another. We provide a detailed example in Chapter 6.

4.1.3 Modularity and Composability

In order to support overlay composition, *Mozlog* supports *Composable Virtual Views* (CViews). These define rule groups that, when executed together, perform a specific functionality.

CView Syntax and Usage

The syntax of CViews is as follows:

`viewName[@locSpec](K1,K2,...,Kn, &R1,&R2,...,&Rm)`

Each CView predicate has an initial set of attributes `K1,K2,...Kn` which are already bound to input values read from another predicate (intuitively, these are like input parameters to a function call). The remaining attributes, `&R1,&R2,...,&Rm`, represent the *return values* from invoking the predicate given the input values. This is akin to the use of input bindings [72] in data integration, which were used to pass data into queriable Web forms to retrieve relation results.

We illustrate using a view definition for the following CView predicate `ping(SrcAddr, DestAddr, &RTT)`:

```
def ping(Src, Dest, &RTT) {
  p1 this.Req@Dest(Src,T) :- this.init(Src,Dest),
      T:=f_now().
  p2 this.Resp@Src(T) :- this.Req(Src,T).
  p3 this.return(RTT) :- this.Resp(T),
      RTT:=f_now()-T.
}
```

Figure 4.1: The ping module in *Mozlog* CView

Any rule that must compute the RTT between two nodes can simply include the `ping` predicate in the rule body. `this` is a keyword used to express the context of the CView. All predicates beginning with `this` are valid only locally within the `ping` CView. There are two new built-in events/actions: `this.init` and `this.return`. Rule `p1`, upon receiving event `this.init` along with the query keys `Src` and `Dest`, takes the current timestamp `T`, and passes the data to the host `Dest` as a ping request. After the destination node receives it in rule `p2`, a ping response event is immediately sent back to the source with the timestamp. In rule `p3`, the source node calculates the round trip time based on the timestamp and issues a `this.return` action that finishes the query processing.

We also list a typical ping implementation in *NDlog* to compare: We note the `ping` CView in

```
p1 pingReq(@Dest,Src,T,E) :- ping(@Src,Dest,E),
    T:=f_now().
p2 pingResp(@Src,Dest,T,E) :- pingReq(@Dest,Src,T,E).
p3 pingResult(@Src,Dest,RTT,E) :- pingResp(@Src,Dest,T,E),
    RTT:=f_now()-T.
```

Figure 4.2: The ping module in *NDlog*

Mozlog written differs from *NDlog* in three aspects: (1) There is a session identifier `E` in *NDlog* to differentiate between different ping requests. *Mozlog* hide it from the protocol specification since it is irrelevant to the ping protocol itself. This improves the readability of the specification. Under the hood, the *Mozlog* compiler automatically translates the CView queries into *NDlog* style rules and applies various kind of techniques to improve performance. The implementation is discussed in detail in Section 4.2.2. (2) By using overlay specifiers, the ping CView is not confined to IP network. Without code modification, it can be used to measure latency between two nodes in any overlay network in Mosaic. (3) The syntax of CView query processing is modular, which further improves readability.

Composition and Resource Sharing

CViews are a natural abstraction for composing control plane functionalities over different overlays. We provide an example to show how to construct trigger sampling in *i*3 by composing Chord and RON. The Chord lookup in CView can be written as:

```
chord.lookup@Ldmk(Key,&DestID,&DestAddr)
```

Given a query on `Key`, it returns the lookup result: the Chord ID of the destination and the network address of the destination. A query with an unbound `Key` will be rejected by the compiler.

RON maintains several CViews to export the current pair-wise EWMA latency, bandwidth and loss rate measurement results. The latency CView is:

```
ron.latency(Src, Dest, &EWMA_RTT)
```

When an *i*3 client tries to locate a private trigger that relays its traffic, it can leverage the RON measurement results and find the best private trigger.

```
/*schema: (Address, Key, RTT) */
materialize(bestPT, SAMPLE_LIFETIME, 1, keys(1), evict max(3)).
```

```
s1 bestPT(KeyAddr, K, RTT) :- periodic(SAMPLE_INTERVAL),
    localAddr(LocalAddr),
    K :=f_randID(),
    chord.lookup@LANDMARK(K, &_, &KeyAddr),
    ron.latency(LocalAddr, KeyAddr, &RTT).
s2 trigger@KeyAddr(NodeID, LocalAddr):-periodic(TRIGGER_REFRESH_INTERVAL),
    node(NodeID), localAddr(LocalAddr),
    bestPT(KeyAddr, _, _).
```

The rules `s1-s2` are used by a local node `LocalAddr` to compute a private trigger with the lowest RTT from itself. Periodically, every `SAMPLE_INTERVAL` seconds, `LocalAddr` picks a random node and obtains a sample RTT. The sampling is performed by rule `s1` using the `chord.lookup` CView predicate to locate a node `KeyAddr` corresponding to a random identifier `K`. Then the `ron.latency` CView predicate obtains the RTT measurements between `LocalAddr` and `KeyAddr`. The use of CViews allows us to perform multiple distributed operations (Chord lookup, followed by RON measurement) all within a single rule. Based on the sampling result stored in `bestPT`, rule `s2` periodically refreshes the current best `trigger` at the node `KeyAddr`.

To summarize, the advantages of CViews are as follows. First, CViews promote code reuse and enable functionality composition between different overlays (as with the shared `ping` CView). Not only is code reused, but network resources are saved. Second, CViews abstract details of asychronous event-driven programming. In the `ping` example, nodes no longer are required to maintain pending state for every ping message that was sent out: the compiler automatically takes care of that. This avoids the tedious churn and failure detection rules often required in original *NDlog* specifications. This enhances readability and makes the code even more concise: the use of CViews reduced the number of lines in Chord by 8 rules (from 43 to 35).

4.1.4 Special Predicates

To interact with legacy applications and provide more transport layer functionalities, *Mozlog* supports several built-in predicates for tun device access and TCP. The `tun` predicate has the following schema: `tun(IPPacket[,SrcIP, DestIP, Protocol,TTL])`. When MOSAIC receives an IP packet from

/dev/net/tun, a tun tuple is injected into the dataflow. IPPacket is the whole IP packet including the header. SrcIP, DestIP, Protocol and TTL are corresponding attributes extracted from the IP header. When tun is an action generated by the rules, IPPacket will be sent to /dev/net/tun. Optionally, the IP header is updated based on the rest of the attributes if given.

We use the following two rules to demonstrate how to use the tun predicate:

```
p2p_tun tun@Peer(Pkt) :- tun(Pkt),
  Peer:="158.130.7.3:1086".
i3_tun tun@Peer(Pkt) :- tun(Pkt, Src, Dest),
  Key:=f_sha1(Dest),
  Peer:=i3::Key.
```

Rule p2p_tun sets up a point-to-point UDP tunnel between the local node and the remote Mosaic node listening at the specific address and port. The peer IP is a constant UDP address. Similarly, rule i3_tun sets up a tunnel via *i*3. It uses the SHA-1 hash of the destination tunneling address as the *i*3 key.

A second set of new predicates is TCP-related: Each predicate corresponds to a system call for TCP sockets. They are tcp.listen, tcp.connect, tcp.accept, tcp.read, tcp.write and tcp.close, provided in the CView syntax. This support provides a foundation for transport layer or session layer overlay [60, 51, 95] support inside Mosaic.

An example use case would be to use tcp.read and tcp.write to forward packets from Skt1 to Skt2.

```
fwdEvent(Skt1, Skt2) :- fwdEvent(Skt1,Skt2),
  tcp.read(Skt1, 0, &Packet),
  tcp.write(Skt2, Packet, &Size).
```

tcp.read has the schema of tcp.read(Skt, Len, &Packet). That is, the query takes a socket descriptor and an integer Len as inputs and returns the actual packet when it is received from the socket. The socket descriptor is obtained from either tcp.accept or tcp.connect. Similarly, tcp.write sends Packet to Skt2. As a recursive rule, it keeps forwarding data packets until one of the sockets is closed.

4.2 Implementation

The Mosaic platform builds on the P2 [55] declarative networking system and adds significant new functionality. The P2 planner and dataflow engine have been revised to generate execution plans that accommodate new language features of *Mozlog*: specifically, those related to runtime support for dynamic location specifier, data plane forwarding, and interactions with legacy applications.

4.2.1 Dataflow

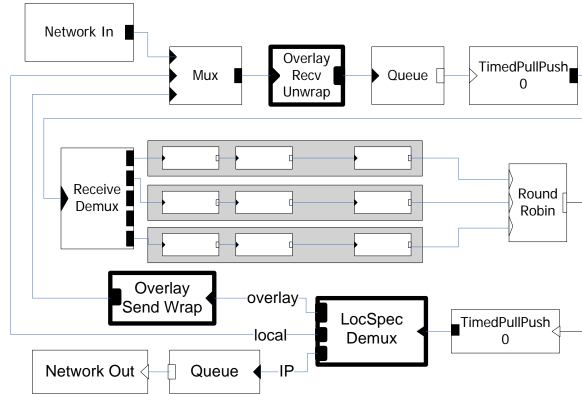

Figure 4.3: System dataflow with dynamic location specifiers.

Figure 4.3 shows a typical execution plan generated by compiling *Mozlog* rules. Similar to P2 dataflows, there are several network processing elements (denoted by `Network In` and `Network Out`) that connect to individual rule strands (inside the gray box) that correspond to compiled database operators. Here, we focus on our modifications, and the interested reader is referred to [55] for details on the dataflow framework.

To implement dynamic location specifiers and overlay forwarding on the data plane, we modify the planner to automatically generate three additional Mosaic elements shown in **bold** in the dataflow: `OverlayRecvUnwrap`, `OverlaySendWrap`, and `LocSpecDemux`. The elements `OverlayRecvUnwrap` and `OverlaySendWrap` are used for de-encapsulation and encapsulation of tuples from overlay traffic.

At the top of the figure, the `Mux` multiplexes incoming tuples received locally or from the network. These tuples are processed by the `OverlayRecvUnwrap` element that will extract the overlay payload for all tuples of the form `overlay.recv(Packet)`, where `Packet` is the payload with type tuple. Since the payload may be encapsulated by multiple headers (for layered overlays), this element needs to

"unwrap" until the payload is retrieved. The `Packet` payload is then used as input to the dataflow via the `ReceiveDemux` element, and used as input to various rule strands for execution.

Executing the rule strands results in the generation of output tuples that are sent to a `LocSpecDemux` element. This element checks the runtime type of the location specifier, and then demultiplexes as follows:

- Tuples `tuplename(F1, F2, ..., Fn)` are local tuples and sent to the `Mux`.
- Tuples `tuplename@IPAddr(F1, F2, ..., Fn)` are treated as regular IP-based tuples and sent to the network directly.
- Tuples `tuplename@ovname::ovaddr(F1, F2, ..., Fn)` are designated for overlay network `ovname` with address `ovaddr`. A new event tuple `ovname.send(ovaddr, tuplename(F1, F2, ..., Fn))` which denotes the `send` primitive of the overlay network `ovname` is generated (see Section 4.1.2). This new tuple is reinserted back to the same dataflow to be forwarded based on the overlay specification.

4.2.2 Compilation of CViews

The *Mozlog*-to-*NDlog* translator requires rewriting and expanding all CView rules into *NDlog* rules, which can then be compiled into dataflow strands using the P2 planner. The compilation process involves a query rewrite that takes as input all CView predicates, and expands them into multiple *NDlog* rules based on their view definitions.

Since this process resembles function call compilation, we reuse the terms *caller* and *callee*. A rule that takes an input CView predicate is the *caller*. The set of rules based on the view definition (*e.g.*, rules `p1-p3` in Section 4.1.3) comprises the operations of the *callee*.

In a typical C compiler, the caller maintains a stack, pushing local variables (execution context) and the return address before a call. Similarly, for each CView input predicate `viewName[@locSpec] (K1,...,Kn, &R1,...,&Rm)`, the execution context is all the bound variables `K1,...,Kn` and the variables that appear in the rule body before the CView term. The expanded rules are executed, and the local variables are stored in a designated internal context table. The local state is stored for the duration of view execution. Each expanded set of rules replaces the `this` prefix in the original view definition with a query context identifier `CID` that uniquely identifies the current invocation

of the view, and a return address `RetAddr` of the caller. When the caller has finished executing all the rules for the view, the results are returned to the caller (`RetAddr`).

We use the `ping` module in Figure 4.1 to demonstrate the CView compilation process.

```
ping_p1 ping_pingReq(@RI, NI, T, CID, LVReturnAddr):-
    ping_init(@NI, RI, CID, LVReturnAddr),
    T := f_now().
ping_p2 ping_pingResp(@RI, T, CID, LVReturnAddr):-
    ping_pingReq(@NI, RI, T, CID, LVReturnAddr).
ping_p3 ping_return(@LVReturnAddr, Delay, CID):-
    ping_pingResp(@NI, T, CID, LVReturnAddr),
    Delay := f_now()- T.
```

Figure 4.4: The result of ping CView translation from *Mozlog* to *NDlog*

Figure 4.4 shows the compilation result from the ping module (Figure 4.1). All predicates within the CView are appended with two fields, `CID` as the query context identifier and `LVReturnAddr` as the return address to the callee.

Suppose the caller rule is

```
r1 pingResult(@NI, RI, Delay):- periodic(@NI, E, 2),
    RI := DESTADDR,
    ping(@NI, RI, &Delay).
```

Rule `r1` periodically measures the RTT to the destination node `RI`. The translation result is showed in Figure 4.5.

```
materialize(r1_ctxt, 1000, 1000, keys(1, 2) ).
r1_cxt r1_ctxt(@NI,CID,E,RI) :- periodic(@NI, E, 2),
    RI := DESTADDR,
    CID := f_rand().
r1_init ping_init( @NI, RI, CID,NI) :- r1_ctxt(@NI,CID,E,RI).
r1_return pingResult(@NI, RI, Delay):- r1_ctxt(@NI, CID, E, RI),
    ping_return(@NI, Delay, CID).
```

Figure 4.5: The result of ping caller translation from *Mozlog* to *NDlog*

First, a materialized table `r1_ctxt` is generated to store query identifiers and query context (bound variables before the CView term) locally. Second, rule `r1_cxt` generates a unique query identifier `CID` and saves the context variables (`NI`, `E`, `RI`). Then, rule `r1_init` invokes the ping CView,

and finally rule `r1_return` takes the return tuple from ping CView, which is joined with the saved context, and emits the result `pingResult`.

4.2.3 Optimizations

We have explored several compiler optimizations. These include *tail recursive CView optimization* to reduce communication overhead, *inline view expansion* by duplicating runtime CView elements to reduce demultiplexing overhead, and *local event shortcut* to shorten the dataflow paths and reduce scheduling overhead.

Tail recursion optimization An interesting optimization opportunity exists when the CView handler recursively queries itself right before it returns (known as tail recursion in programming languages).

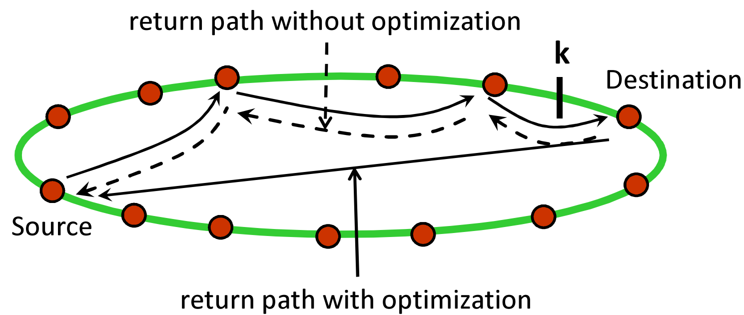

Figure 4.6: An illustration of tail recursive optimization on Chord lookup.

The lookup portion of the Chord DHT protocol implementation (Figure 4.7) recursively queries the finger nodes until the destination is reached, as showed evidently in rule `l4`, where the last query term in the predicate `this.return` is `chord_lookup` itself.

The *Mozlog* compiler automatically translates such calls so that the destination node can send the query results back directly to the query initiator, rather than let the result tuple traverse the entire call path. To achieve this, the return node address (i.e. the source node) and the original `CID` are passed along with each `chord.lookup` query. This is illustrated in Figure 4.6. As a result, tail recursion optimization reduces the Chord lookup latency.

Local dataflow optimization When a CView only involves non-recursive, local query terms, the dataflow can further be optimized at compile time. Syntactically, local CViews are equivalent

```
def chord_lookup(@NI, K, &S, &SI){
l1 this.return(S,SI) :- this.init(@NI,K),
    node(@NI,N),
    bestSucc(@NI,S,SI),
    K in (N,S].
l2 this.bestLookupDist(@NI,K,a_MIN<D>) :- this.init(@NI,K),
    node(@NI,N),
    finger(@NI,I,B,BI),
    D := K - B - 1,
    B in (N,K).
l3 this.forwardLookup(@NI, a_MIN<BI>,K) :- node(@NI,N),
    this.bestLookupDist(@NI,K,D),
    finger(@NI,I,B,BI),
    D == K - B - 1,
    B in (N,K).
l4 this.return( S, SI) :- this.forwardLookup(@NI, BI, K),
    f_typeOf(BI) != "null",
    chord_lookup(@BI, K, &S, &SI).
}
```

Figure 4.7: Chord lookup in CView

to function calls in imperative languages, such as C: the bound variables are the function inputs and unbound variables are the outputs. Therefore, similar to inline function expansion, we can also expand local CViews. Without inline expansion, every CView rule maps to a rule strand in the dataflow. With inline expansion, every rule may have multiple instances of the rule strands. Each instance belongs to a rule that has the CView term in its rule body. The optimization allows to avoid using temporary tables to save execution context (*i.e.* no call stacks are saved). We use this optimization for all TCP related predicates and have observed dramatic performance improvement (See Section 5.6.1).

4.2.4 Special Predicates

The `tun`-, and `tcp`-related special predicates are treated differently from ordinary tuples in the dataflow by the planner. Each special predicate has a rule strand in the dataflow, between the `ReceiveDemux` element and the `RoundRobin` element(see Figure 4.3).

The `tun` tuples are mostly used in setting up tunnels to provide network layer legacy support in Section 4.2.5. In the dataflow, two elements `Tun::Tx` and `Tun::Rx` are inserted in the `tun` rule strand right after `ReceiveDemux`. `Tun::Rx` reads IP packets from the tun device, generates the `tun`

tuple, and sends to the next element in the rule strand; `Tun::Tx` receives a `tun` tuple, formats it to an IP packet and writes to the tun device.

The TCP related predicates are used for creating transport layer legacy support (see Section 4.2.5). Each TCP predicate has a corresponding input and output event handler. To use `tcp.read` as an example, first the CView compiler translates each rule that contains `tcp.read` to the *NDlog* format, which generates a `tcp.read.init` event and waits for the `tcp.read.return` event. The `tcp.read.init` event is connected to the `tcpRead` element. It adds the socket descriptor to the select() pool of the P2 event loop. Once data is available, the P2 event loop calls back to the element, which then removes the socket descriptor from the select() pool, reads the packet, and sends a `tcp.read.return` event tuple containing the packet.

4.2.5 Legacy Support

MOSAIC adopts two mechanisms to support legacy applications at different layers. At the network layer, we use the tun device to provide overlay tunnels between legacy applications. For each end host, it takes a private IP address from 1.0.0.0/8 to avoid conflict from other public IP networks. After a legacy application sends a packet to an address in the tun network, the kernel redirects it to MOSAIC, which generates a `tun` tuple. Currently there is an address translation rule to use a special mapping table to translate the private IP address to the overlay address. This can be extended to use any name resolution service in the future by combining DNS request hijacking [42]. After address translation, the packet tunneling rules we described in Section 4.1.4 deliver the IP packet to the destination via the corresponding overlays.

To use IP layer overlays, such as *i*3 or RON, IP tunneling is mandatory. For transport layer overlays that intend to replace or augment TCP, we provide an alternative way to leverage the dynamic library call interceptions. The environment variable LD_PRELOAD is set to our customized library to intercept the socket system calls at the user space. Compared to the tun approach at the IP layer, library interceptions avoid the overhead of an extra memory copy between the kernel and user space, and expose the connection oriented primitives from the application to the transport overlays. Based on the TCP predicates, we have implemented a SOCKS proxy [49] in our prototype, which can be viewed as a two-hop transport layer overlay that does source routing to traverse firewall.

To support a legacy overlay that is not implemented in MOSAIC, we build an adapter for the

overlay to interact with MOSAIC via the `send` and `recv` primitives. The adapter redirects `legacy.send` tuple from the dataflow to the overlay, and inject `legacy.recv` tuple upon overlay's packet reception. Because the legacy overlays are built on IP, they can only be bridged with other overlays or used as substrates underneath other networks, but cannot be layered on top of another overlay for either the control or the data plane.

4.3 Summary

In this chapter, we presented the design and implementation of the *Mozlog* language. Our language is based on *NDlog*, and we described the extension features including *flexible naming and addressing*, *data and control plane integration*, *modularity and reusability*, and *special predicates*. All these extensions are motivated by the requirement from the MOSAIC architecture to provide a extensible, composable and practical network architecture. In the following two chapters, we present two concrete ways to extend networks, and describe in detail how *Mozlog* programs are used and executed to implement the network protocols.

Chapter 5

Programming Network Services

In this chapter, we describe how to introduce new network protocols in Mosaic by programming them in *Mozlog*. We provide several example classes of useful queries in support of reliability, rendezvous-based communication and mobility (*proxy location*, *customizable routing*, and *service discovery*). This is not intended to be an exhaustive coverage of all the possibilities of application scenarios, but an illustration of the ease with which *Mozlog* queries can be used for implementing network services that can be easily composed and enhanced for various aspects.

5.1 Resilient Overlay Network

Our first example (rules `d1-d5`) shows a declarative implementation of Resilient Overlay Network (RON) [7] over an existing IP infrastructure. RON enables routing around failures, where packets can be sent via overlay relay nodes that bypass failures in the underlay. Each node maintains a list of all other participating RON nodes (stored as `ron_nodes` table). At each node `NI`, rule `d1` periodically measures the round-trip time, and propagates this information as `link_delay` tuples to all other RON nodes. The predicate `ping(@NI,R,&RTT)` is a CView which consists of a set of rules that when executed, returns the RTT from `NI` to `R`. Based on the `link_delay` measurements, each node computes one-hop and two-hop paths using rules `d2-d3` respectively. Computations stop at two hops since a single overlay relay is sufficient to route around most failures. We further added a `0.1s` penalty for utilizing a 2-hop indirection. Rules `d4-d5` are then used to select the path with the shortest cost for any given `NI` to `D`. In rule `d4`, `min` is an aggregate, and the rule will compute

```
#define PERIOD 30
d1 link_delay(@RI,NI,R,RTT) :- periodic(@NI,E,PERIOD),
    ron_nodes(@NI,R), R!=NI,
    ping(@NI,R,&RTT),
    ron_nodes(@NI,RI).
d2 path(@NI,D,D,RTT) :- link_delay(@NI,S,D,RTT),
    S==NI.
d3 path(@NI,D,N,C) :- link_delay(@NI,NI,N,RTT1),
    link_delay(@NI,N,D,RTT2),
    N!=NI,
    D!=NI,
    N!=D,
    C=RTT1+RTT2+0.1.
d4 bestPathCost(@NI,D,min<C>) :- path(@NI,D,N,C).
d5 bestPath(@NI,D,N,C):- bestPathCost(@NI,D,C),
    path(@NI,D,N,C).
```

Figure 5.1: RON control plane *Mozlog* program

the minimum `C` for each (`NI`, `D`) pair. Rule `d5` computes the actual best paths, and the `bestPath` table stored at each node `NI` can then be used to route packets over the RON overlay to node `N` along the shortest path to `D`.

5.2 Internet Indirection Infrastructure

In this example, we demonstrate how to implement Internet indirection infrastructure (*i*3) [84] in *Mozlog*. *i*3 is a rendezvous-based overlay network that provides a level of indirection between senders and receivers. In *i*3, instead of explicitly sending a packet to a destination, each packet is associated with an identifier; this identifier is then used by the receiver to achieve delivery of the packet. This indirection mechanism is implemented with the use of *triggers*. The *i*3 infrastructure nodes stores triggers as soft state, and the edge nodes periodically maintain their trigger state on the infrastructure nodes. In our system, each `trigger(@PI,P,NI)` condition is encoded in a tuple, which is stored at the infrastructure node `PI`; it specifies to route messages on behalf of mobile host `NI` with identifier `P`.

The control plane program is showed in Figure 5.2. Two CViews `i3_lookupNoCache` and `i3_lookup` provide trigger lookup function in *i*3. They are built upon the CView `chord_lookup` from Chord DHT [85]. The chord lookup request is either sent to itself if the node is an infrastructure node (rule `l1i`) or a landmark node otherwise (rule `l1`). In rule `l3`, the lookup result is cached in a

```
materialize(triggerCache, 180, 100, keys(2)).

/*input K: key. output: KI: IP that handles key K*/
def i3_lookupNoCache(@NI, K, &KI) {
#ifdef INFRASTRUCTURE
l1i this.lookupResults(@NI, K, S, SI) :- this.init(@NI, K),
    chord_lookup(@NI, K, &S, &SI). /*call chord lookup locally*/
#else
l1 this.lookupResults(@NI, K, S, SI) :- this.init(@NI, K),
    i3_landmark(@NI, LI),  /*call chord lookup through landmark*/
    chord_lookup(@LI, K, &S, &SI).
#endif
l2 this.return(KI) :- this.lookupResults(@NI, K, S, KI).
l3 triggerCache(@NI, K, KI) :- this.lookupResults(@NI, K, S, KI).
}
def i3_lookup(@NI, K, &KI) {
l0  this.cachedResult(@NI, K, min<KI>) :- this.init(@NI, K),
    triggerCache(@NI, K, KI).
l01 this.return(KI)  :- this.cachedResult(@NI, K, KI1),
    f_typeOf(KI1) == "null",
    i3_lookupNoCache(@NI, K, &KI).
l02 this.return(KI) :- this.cachedResult(@NI, K, KI),
    f_typeOf(KI) != "null".
}
l4 trigger(@KI, K, NI) :- periodic(@NI, E, TRIGGER_REFRESH_RATE),
    node(@NI, K),
    i3_lookupNoCache(@NI, K, &KI).
```

Figure 5.2: *i*3 control plane *Mozlog* program

table `triggerCache`. In CView `i3_lookup`, it first examines if the key `K` is in `triggerCache` (rule `l0`. If so, cached result is directly returned without generating network traffic (rule `l02`. Otherwise, the queries in `i3_lookupNoCache` are executed (rule `l01`). For an edge node, it periodically issues *i*3 lookup queries using their own node id `K` as the key, and update its trigger which is stored at network address `KI` (rule `l4`).

5.3 Proxy Location

In this example, a mobile host issues an *Mozlog* query to locate a nearby infrastructure node that serves as its proxy node either for routing data, or for providing support for location-based services. The examples are based on ROAM [99] and DHARMA [60].

5.3.1 DHARMA

DHARMA [60] can also be used to select a nearby proxy node. Unlike ROAM's DHT, DHARMA uses a number of designated *portal servers*, nodes that store information on other nodes in the infrastructure.

```
#define R_RATE 5
d1 agentMsg(@PI,NI) :- periodic(@NI,E,R_RATE),
    portal(@NI,PI).
d2 pingResp(@RI,NI,E) :- pingReq(@NI,RI,E).
```

We consider a simple example with one portal server `PI`, stored on every infrastructure node `NI` as a `portal(@NI,PI)` tuple. Rule `d1` is executed at all infrastructure nodes, and will result in the generation of periodic heartbeats (`agentMsg(@PI,NI)` tuples) to the portal server `PI`. Rule `d2` generates a `pingResp` in response to a ping request.

```
d3 agentList(@PI,NI) :- agentMsg(@PI,NI).
d4 agentCandidates(@MI,AI,E) :-
    requestProxy(@PI,MI,S,E),
    agentList(@PI,AI), f_coinFlip(S)=1.
```

Rules `d3-d4` are executed on the portal server. Rule `d3` stores all incoming `agentMsg(@PI,NI)` tuples in a `agentList(@PI,NI)` table, hence maintaining the list of all candidate infrastructure nodes `NI`. These tuples will timeout unless the respective infrastructure nodes periodically refresh their entries via regular `agentMsg` messages. In rule `d4`, a `agentCandidates` tuple are generated in response to a mobile host request for candidate proxies (via a `requestProxy` tuple).

```
#define P_INTERVAL 5
#define S_RATE 0.2
d5 requestProxy(@PI,NI,S_RATE,E) :-
    periodic(@NI,E,P_INTERVAL),
    portal(@NI,PI).
d6 pendingPing(@NI,AI,E,T) :-
    agentCandidates(@NI,AI,E),
```

```
    T=f_now().
d7 pingReq(@AI, NI, E) :- agentCandidates(@NI,AI,E).
d8 pingRTT(@NI,SI,RTT) :- pingResp(@NI,SI,E),
    pendingPing(@NI,SI,E,T), RTT=f_now()-T.
d9 leastRTT(@NI,E,min<RTT>) :-
    periodic(@NI,E,P_INTERVAL),
    pingRTT(@NI,RI,RTT).
d10 leastRTTNode(@NI,RI,RTT) :- leastRTT(@NI,E,RTT),
    pingRTT(@NI,RI,RTT).
d11 proxy(@NI,RI) :- leastRTT(@NI,E,RTT),
    pingRTT(@NI,RI,RTT).
Query proxy(@NI,RI).
```

The above rules `d5-d11` are executed by a mobile host seeking to locate a nearby proxy node. Rule `d5` results in the mobile host periodically generating a `requestProxy` tuple to the portal server, which will return 20% (determined by `S_RATE`) of the nodes from its `agentList` table as potential proxy candidates. Similar to the earlier ROAM rules, `d6-d11` computes the closest proxy node `RI`, which is maintained at the mobile host `NI`.

5.3.2 ROAM

ROAM is built on top of *i*3 [84]. The following rules `i1,i2` are issued by a mobile host for selecting the closest infrastructure node as its proxy.

```
#define SIZE 5
#define S_RATE 60
/*schema: @NI, RTT, KEY, KEYIP; evict policy: max*/
materialize(bestProxy, SAMPLE_LIFETIME, 1, keys(1,4),evict max).
i1 randomKeys(@NI,K) :- periodic(@NI,E,0,SIZE),
    K=f_randID().
i2 bestProxy(@NI, RTT, K, KI) :- periodic(@NI,E1,S_RATE),
    randomKeys(@NI,K),
    landmark(@NI, LI),
```

```
    i3_lookup(@LI, K, &KI),
    ping(@NI, KI, &RTT).

Query bestProxy(@NI,RTT, K, KI).
```

The *i*3 overlay utilizes a DHT to provide the mapping from identifiers to hosts. Our rules assume that the declarative *i*3 *Mozlog* code as described in Section 5.2 is executed on all infrastructure nodes.

Rule `i1` periodically generates `SIZE` number of random keys stored in `randomKeys` table using the built-in function `f_randID` that will return a 160-bit random identifier. Every `S_rate` seconds, rule `i2` generates lookup requests from the mobile host `NI` to its landmark node `LI` using CView `i3_lookup`, one for each random key `K` generated in rule `i1`. After retrieving lookup results from the *i*3 landmark, the second CView in the rule `ping` is triggered to send a RTT measurement request to the node with address `KI`, and store the result in the `bestProxy` table. Note that in the `materialize` statement, table `bestProxy` is defined to have at most 1 row. When exceeding the limit, an eviction policy is used to evict the tuple with the maximum value. Therefore, each time a new tuple is inserted to the table, the one with the larger RTT will be evicted. As a result, table `bestProxy` holds the proxy with the minimum RTT.

Note that even though ROAM is more complicated than DHARMA in terms of proxy selection functionality, the rules are substantially simpler. This is because in the DHARMA proxy location program, we intentionally did not use CViews or the table eviction feature. Therefore, we have to rewrite `ping` related rules and the aggregate queries. On the other hand, the *i*3 proxy location program can effectively reuse relevant code in CViews.

5.3.3 Flexible Proxy Selection

The *Mozlog* query language enables higher-level concepts to be easily encoded by making minor modifications to the above rules. This enables user-customizable proxy selections. For example, by replacing `min` with `min-k`, we can select the *top-k* nodes to get multiple proxies per mobile host. We can also adopt different criteria: instead of selecting the closest RTT node, we can select the least loaded node as long as it is within a RTT bound. We can also limit our proxy selection to nodes that provide certain services within their location (e.g. transcoding services described in Section

5.5).

5.4 Customizable Routing

Our next example shows a customizable version of the basic path vector protocol [56]. The query computes the best paths among all infrastructure nodes. The query takes as input `link(@S,D,C)` tuples, where each link from node `S` to node `D` denotes connectivity between two infrastructure nodes; messages can be routed from `S` to `D` at cost `C`.

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

Rules `bp1` and `bp2` compute all possible paths, and rules `bp3` and `bp4` compute all-pairs best paths, which are stored as `bestPath` tuples at each source node `S` for source routing. We have left the aggregation function `AGG` unspecified. By changing `AGG` and the function `f_compute` used for computing the path cost C, the above query can generate best paths based on any metric including link latency, loss rates, available bandwidth and node load. For example, if the query is used for computing the shortest paths, `f_sum` is the appropriate replacement for `f_compute`, and `min` is the replacement for `AGG`. The above query can be further restricted by the current sender and receiver proxies of the communicating devices, and the routes between these two proxies is maintained as a continuous query, and adapted based on link updates.

We can extend the query by adding constraints based on the session requirements, by introducing an additional `session` predicate to the rules above. For example, we can restrict the set of paths to those with costs below a loss or latency threshold K by adding a `session(@S,K)` predicate, and a constraint `C<K` to the rules `bp1` and `bp2`.

5.5 Service Discovery

The proxy location query described in Section 5.3 is an instance of service discovery, in which a nearby routing proxy is located. Once a proxy node is identified, a mobile host can issue additional queries via its proxy to locate desired resources within its vicinity. In addition, the use of a declarative framework eases the composition of services. For example, one can query for multiple services within the infrastructure, and then construct a path (either an explicit path presented Section 5.4, or a series of triggers supported by *i*3) along all intermediate service points.

In this example, we build upon the earlier ROAM example, to demonstrate service discovery and composition with the use of *i*3 triggers. Here a sender `SI` performs a discovery of a transcoder, and then forwards all packets to the transcoder before being delivered to their receivers `RI`. Our example is presented with flexibility and composability of our infrastructure in mind. While our example is based on *i*3 and ROAM, our infrastructure does not preclude supporting other discovery and composition mechanisms.

```
t1 leastLoad(@PI,SI,min<L>) :-  proxy(@SI,P,PI),
    transcoders(@PI,TI,TID,L).
t2 bestTranscoder(@SI,TI,TID) :- transcoders(@PI,TI,TID,L),
    leastLoad(@PI,SI,L).
Query bestTranscoder(@SI,TI,TID).
```

Each transcoder `TI` inserts a trigger, stored as a `trigger(@PI,TID,TI)` tuple at the infrastructure node `PI` that owns the identifier `TID`. These triggers are further registered in the `transcoders` table of nearby proxy nodes. The rules `t1-t2` are used by a sender `SI` to locate a least loaded transcoder `TI` with identifier `TID` registered at `S`'s proxy `PI`.

When a sender `SI` wishes to send a packet to a mobile host `NI` supported by a trigger with identifier `RID`, a series of identifiers `(TID,RID)` is required to first forward each packet to the transcoder (`TID`), which then reroutes the packet to the mobile host (`RID`). We omit the (few) *Mozlog* rules for forwarding with triggers. The main takeaway is that *i*3's service composition mechanism via multiple triggers can be easily supported by our infrastructure. In addition, we can use the declarative interface to query and locate the triggers themselves during service discovery.

As a further enhancement, we can enhance existing queries to support *late binding* [3]. An

Mozlog query can be sent along with each packet, which will be routed based on the service requests indicated in the *Mozlog* query. Support for late bindings require additional *Mozlog* rules for publishing and propagating service descriptions among infrastructure nodes to create routing tables based on service attributes. This provides a flexible data delivery mechanism that allows applications to track rapid change (e.g., user or network mobility) and support changing service updates.

5.6 Evaluation

In this section, we present the evaluation of Mosaic on a local cluster and on Emulab. First, we validate that *Mozlog* specifications for declarative networks, tunneling and packet forwarding are comparable in performance to native implementations. Second, we use our implementation to demonstrate feasibility and functionality, using actual legacy applications that run unmodified on various composed overlays using Mosaic. Third, we evaluate the dynamic composition capabilities of Mosaic.

In all our experiments, we make use of a declarative Chord implementation which consists of 35 rules. Our *i*3 implementation uses Chord and adds 16 further rules. We also implement the RON overlay in 11 rules. Both *i*3 and RON can be used by legacy applications via the `tun` device, as described in Section 4.1.4.

5.6.1 LAN Experiments

To study the overhead of Mosaic, we measured the latency and TCP throughput between two overlay clients within the same LAN. The experiment setup was on a local cluster with eight Pentium IV 2.8GHz PCs with 2GB RAM running Fedora Core 6 with kernel version 2.6.20, which are interconnected by high-speed Gigabit Ethernet. While the local LAN setup and workload is not typical of Mosaic's usage, it allows us to eliminate wide-area dynamic artifacts that may affect the measurements. We measured the latency using `ping` and TCP throughput using `iperf`.

Network Layer Overlay Overhead

In the experiments, we use the tun device to provide legacy application support for network layer overlays. MTU was reduced to 1250 bytes to avoid fragmentation when headers were added. The

test	latency(ms)	throughput (KByte/s)
DirectIP	0.10	97994
OpenVPN	0.30	13951
MozTun	0.50	8353
RON	0.71	5796
*i*3	1.31	3299

Table 5.1: Overhead comparison in LAN

measurement results are shown in Table 5.1 for the following test configurations:

- **DirectIP:** Two nodes communicate via direct IP, where `iperf` can fully utilize the bandwidth of the Gigabit network. This serves as an indication of the best latency and throughput achievable in our LAN.

- **OpenVPN:** OpenVPN [94] 2.0.9 is a widely used tunneling software. We set up a point-to-point tunnel via UDP between two cluster nodes and disabled encryption and compression. The performance results provide a baseline for the overhead using the *tun* device virtualization. Compared to DirectIP, the latency increases by around $0.2ms$, and the TCP throughput drops by a factor of more than 6. This overhead is inevitable for all overlay networks supporting legacy applications using the *tun* device, including those hosted on MOSAIC.

- **MozTun:** We set up a static point-to-point tunnel in MOSAIC between two cluster nodes using the following rule:

  ```
  tun@PeerIP(Pkt) :- tun(Pkt), PeerIP:=PEERADDR.
  ```

  MozTun and OpenVPN essentially have the same functionality except that MozTun is implemented in MOSAIC. The additional overheads in throughput and latency are solely attributed to the MOSAIC dataflow processing overhead bounded by CPU capacity. In MozTun, the latency increased $0.20ms$ over OpenVPN, which is negligible when executed over wide-area networks.

- **RON:** We ran the RON network using MOSAIC and utilize two nodes to run the measurements. Since RON does not provide any benefit in our LAN setting with no failures, the comparison to MozTun is used to show the extra overhead for rule processing in our implementation.

- **i3:** Six nodes were set up as *i*3 servers, using Chord to provide lookup functionality. The remaining two nodes were selected as *i*3 clients. A packet sent by the source *i*3 client to the

	TCP	SOCKS unoptimized	SOCKS optimized
throughput (KB/s)	97994	8132	97186

Table 5.2: Overhead comparison in LAN between native TCP, SOCKS proxy in Mosaic, and SOCKS proxy in Mosaic with optimized dataflow

destination *i*3 client went through the public trigger of the destination, which was hosted on the *i*3 server of another cluster node. Since it introduced a level of indirection plus extra rule processing overhead, *i*3 added the most cost among the 5 configurations studied.

In summary, the overhead of Mosaic is respectable: the throughput of Mosaic's point-to-point tunneling (MozTun) is comparable to that obtained by using well-known tunneling software (OpenVPN). In the extreme case (level of indirection of *i*3 with tunneling), the extra latency ($1.2ms$) incurred is negligible for an application running on wide-area networks.

Transport Layer Overlay Overhead

Our proof-of-concept implementation of a transport layer overlay is a SOCKS proxy using 18 *Mozlog* rules. The SOCKS protocol [49] is a transport layer protocol, which can be viewed as a transport-layer overlay network with one level of indirection for firewall traversal. We used the library interception technique mentioned in Section 4.2.5 to support legacy TCP applications.

We deployed our SOCKS proxy on the client and used `iperf` to measure TCP throughput between the client and the server. From the measurement results in Table 5.2, we observe that by using a different virtualization technique, the SOCKS proxy achieves better throughput than OpenVPN in Section 5.6.1. In addition, by applying *inline view expansion* and *local event shortcut* optimizations as described in Section 4.2.3, the throughput increases dramatically and approaches that of native TCP. The performance improvement obtained by our SOCKS proxy suggests that Mosaic is able to translate and optimize high-level declarative specifications to efficient implementations for the data plane. A detailed performance study on the optimization is outside the scope of this dissertation and is a subject of future work.

5.6.2 Emulab Experiments

In this section, we present an evaluation of our proposed infrastructure using the Mosaic system on the Emulab [20] testbed. We focus our evaluation on the proxy location queries (see Section 5.3)

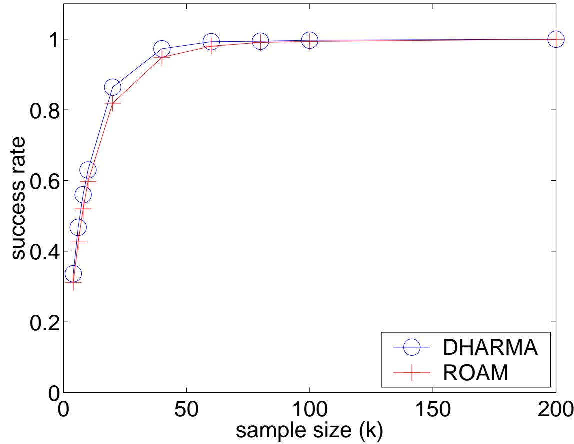

Figure 5.3: Success rate vs sample size for proxy location using DHARMA and ROAM.

for ROAM and DHARMA. Our experimental setup on Emulab consists of 50 nodes organized into 11 routing domains interconnected in a star topology. On each Emulab node, we run 4 MOSAIC processes, for 200 emulated nodes in total.

In our first experiment, we execute the rules of Section 5.3.1 that implement the DHARMA overlay network. We first randomly pick one node to be the portal server maintaining a list of current live home agents. The remaining nodes then periodically report their status to the portal server. In steady state, we randomly pick one of the Emulab nodes as a mobile host. This mobile host issues a query to the portal server, which triggers additional rules (see Section 5.3.1) to select k nodes as samples, and then locates the closest node (in terms of network latency) from these sample nodes as the proxy. If the chosen proxy is within the same routing domain as the mobile host, the proxy location "succeeds."

Figure 5.3 shows the success rate of DHARMA as the sample size increases. For any given sample size, we execute the proxy selection query from 1000 randomly selected locations in the network. As expected, when the sample size increases, the success rate increases. When the sample size is as large as the number of infrastructure nodes (200), all nodes are sampled, and hence the closest proxy is guaranteed to be within the domain. Sampling only 10% of the nodes, DHARMA achieves a success rate of 85%.

In our second experiment, we execute Declare-Chord on all infrastructure nodes for proxy selection. After starting a 200-node Chord network, each mobile host executes the proxy selection queries (as described in Section 5.3.2) by repeatedly sampling the infrastructure nodes via Chord

lookups. The closest node in network distance is then selected as the proxy. Figure 5.3 shows the corresponding success rate of ROAM as the sample size increases. As before, we iterate the proxy selection process from 1000 randomly selected network locations. ROAM's performance is roughly that of DHARMA. When the sample size is 20 nodes, the probability of picking a close proxy is 81%. ROAM has a slightly worse success rate than DHARMA for small k, because the node identifiers are not uniformly distributed for the small 200-node Chord network. As the number of Chord nodes increases, the identifier space will be more evenly distributed, and the performance of ROAM will approach DHARMA. ROAM has the advantage that it avoids the use of a centralized portal server.

The results on Emulab show that we can implement DHARMA and ROAM to perform effective location of proxies that map to nearby locations. As shown in Section 5.3, these specifications can be written in a few *Mozlog* rules each, significantly easing the process of deploying new mobility-based solutions. Comparisons between DHARMA and ROAM are not the point of this dissertation, rather our experiences with two suggests that our infrastructure can be used to rapidly develop and deploy multiple concurrent mobility-based schemes.

5.7 Summary

In this chapter, we provided a portfolio of concrete examples of *Mozlog* programs to demonstrate how to introduce new network services in Mosaic. The examples include a reliability overlay (RON), rendezvous-based communication service (*i*3), and several mobility-related services such as proxy location, customizable routing and service discovery. We then used a local cluster and Emulab as testbeds to evaluate the performance of the services hosting on our Mosaic platform, and showed that the performance is respectable compared to native implementations. In the next chapter, we will describe the second method to extend networks in Mosaic by composing existing network services.

Chapter 6

Composing Network Services

In this chapter, we demonstrate Mosaic's ability to create new network services based on existing network compositions including bridging, layering and hybrid compositions.

6.1 Compiling Compositions

This section describes how the Mosaic compiler automatically translates specifications into *Mozlog* rules. We first define the following reserved tables at each node, which are used in the composition process later:

- `netAddress(@N,OID,Addr)` tracks all current addresses `Addr` of the overlays `OID` in which the node participates at node `N`. If a node has a publicly reachable IP address, a default entry is added as `(0,current_ip)`, where 0 is a reserved ID for the Internet. `OID` can also refer to a bridged network, in which case `Addr` can refer to a source routing address (See Section 6.1.3). Other overlay specific addresses are maintained by the corresponding overlay modules.

- `underlay(@N,OID,Addr)` is used in layering. It stores the mapping from an overlay's `OID` to its current underlay's runtime address `Addr` at node `N` for each deployed overlay. By updating this table, one can switch the underlay being used.

- `forward(@N,OID,Addr)` is used in bridging. It specifies that all packets designated for overlay `OID` are to be sent to the designated gateway with address `Addr`.

Algorithm 6.1 Pseudo code for composition process

```
1  input: spec as the composition specification
2
3  for m in spec.composition.modules:
4      if m.style="Extend":
5          //import the existing nodelist from the directory service
6          m.nodelist += query_directory(m.oid, "nodelist")
7  //check constraints for bridging
8  for l in spec.composition.links:
9      match l with
10     "Bridging" first, second "via" gw:
11         //bridging the first network with the second via the gateway
12         assert(gw ∈ first.nodelist and gw ∈ second.nodelist)
13     |"Bridging" first, second:
14         //gateway is not specified, find default gateway
15         if first.defaultgw ≠ None:
16             l.gw = first.defaultgw
17         else if second.defaultgw ≠ None:
18             l.gw = second.defaultgw
19         else:
20             raise Exception("cannot set gateway address")
21         assert(l.gw ∈ first.nodelist and l.gw ∈ second.nodelist)
22 //check constraints for layering
23 for l in spec.composition.links:
24     match l with
25     "Layering" top "over" bottom:
26         if not top.nodelist ⊆ bottom.nodelist:
27             raise Exception("node sets are wrong for layering")
28 compatibility_check(spec.composition.links)
29 for l in spec.composition.links:
30     match l with
31     "Layering" top "over" bottom:
32         layering(top, bottom)
33     "Bridging" first, second "via" gw "as" bridgename:
34         bridging(first,second,gw,bridgename)
```

6.1.1 Compilation Steps

To create an overlay network composition from scratch, the MOSAIC compiler takes as input a composition specification as presented in Section 3.3.1, and then generate *Mozlog* rules that bridge and layer the appropriate overlay modules. The pseudo code is shown in Algorithm 6.1. In particular, the following steps are taken:

- Line 3-6: compute the *node membership sets* to which each overlay module is to be deployed. If a module is extended from an existing overlay, import existing node list by querying the

directory service (in function `query_directory`) using the oid of that module.

- Line 8-21: check that any bridging gateway node is shared by both networks to be bridged. We use a pattern matching syntax to examine and "extract" information from each link l. When a bridging link specification includes the default gateway (Line 10), we extract the two networks to be bridged as $first$ and $second$, and the gateway as gw; when the bridging link does not include the gateway (Line 13), we first set the gateway to either the default gateway of the first network (Line 15-16) or the default gateway of the second network (Line 17-18), or raise an exception when neither exists (Line 20).

- Line 23-27: check that in each layering configuration, the overlay nodes are also members of the underlay network. Again, we use the pattern matching syntax to extract the overlay as *top* and the underlay as *bottom*.

- Line 28: validate the compatibility of the composition specification. Reject those that may causes problems in feature interaction. (See Algorithm 6.4 in Section 6.1.4)

- Line 31-32: for each overlay layered over another module, add mappings binding each node's logical address in the current overlay to a lower-level underlay address in the `underlay` table. The `layering` function is described in Algorithm 6.2. (Section 6.1.2.)

- Line 33-34: for each overlay module with a bridge, based on the specification, add pre-configured forwarding state entries in the `forward` table or on-demand source routing rules to all member nodes, specifying either the static address of each bridged network's gateway node or the anycast address with each bridged network's ID. The `bridging` function is described in Algorithm 6.3. (Section 6.1.3.)

After the compilation, the rules are shipped to the corresponding physical nodes for deployment.

To modify an existing network composition, most of the procedure remains the same except that the node membership sets of existing overlays are obtained by querying the directory service, and modified *Mozlog* rules are uploaded to the physical nodes to implement the new composition.

6.1.2 Layering

Layering of a control or data plane over another overlay's data plane is achieved through the use of the `underlay` table describing bindings from each overlay node to its current runtime underlay

Algorithm 6.2 pseudo code for layering-related rule generation

```
1  function layering(top, bottom):
2  input: network top, network bottom //layer top over bottom
3  output: rules for layer bindings
4  for n in top.nodelist:
5      if top.oid ∉ n.deployed: // network top is not deployed on node n
6          code = fetch(top.codeurl)
7          addRules(n, code)
8          addRules(n, "underlay(@n, top.oid, Addr) :- netAddress(@n, bottom.oid,
   Addr).")
9      else:
10         updateRules(n, "underlay(@n, top.oid, Addr) :- netAddress(@n, bottom.oid,
   Addr)")
```

address. Abstracting the bindings into a table provides a simple mechanism for switching overlays: Mosaic can simply update the `underlay` table — changing both the underlay protocol and node address as appropriate.

Given a composition specification with layering links, *Mozlog* rules are generated to implement the layering. The pseudo code for rule generation is shown in Algorithm 6.2. If the overlay *top* is not deployed on the node n, we first add the overlay *Mozlog* implementation to the code to be deployed on node n (Line 6-7), then bind the address of network *bottom* to the underlay table entry that belongs to network *top* (Line 8). Note that symbols in italic fonts, including n, *top.oid*, and *bottom.oid* are constants when added into the rule. If *top* is deployed, then there should be a rule that binds its `underlay` table already. We update that rule to the new binding (Line 10).

We illustrate using an example where there are two RON overlays, layered over IP and $i3$. Based on the specifications, at every node, there are two instances of RON executing (`ron_oid1` and `ron_oid2`), and one instance of $i3$ (`i3_oid`). The following *Mozlog* rules `b1` and `b2` are generated to build the two networks:

```
b1 underlay(@N,ron_oid1,U):-netAddress(@N,0,U).
b2 underlay(@N,ron_oid2,U):-netAddress(@N,i3_oid,U).
```

Since `ron_oid1` utilizes IP for routing, rule `b1` takes as input `netAddress(@N,0,U)`, based on the executing node's default IP address. On the other hand, `ron_oid2` routes over $i3$, hence its underlay tuple stores the address of the underlying `i3_oid` node retrieved from the local `netAddress` table.

Note that the layering association is not static. A deployed, running overlay network can switch the underlying network from one to another by updating its underlay table entries at runtime.

This enables dynamic overlay composition. We will discuss an example of dynamic switching in Section 6.2.

Next, the rule to update the `netAddress` table is generated for the newly created overlay. Because the rule is overlay specific, it is not automatically generated by the compiler, but provided by the overlay programmers. For example, consider the *i*3 and RON overlays with identifiers `i3_oid` and `ron_oid` respectively. In *i*3, its overlay address is the SHA-1 hash of the node's public key `K` (as shown in rule `d1`).

```
d1 netAddress(@N,i3_oid, A) :- publicKey(@N,K),
    A:=i3_oid::f_sha1(K).
```

On the other hand, in RON, since its routable address is tightly coupled with its underlay, its address is its own underlay address (typically the IP address that RON uses) annotated with the overlay id as shown rule `d2`:

```
d2 netAddress(@N,ron_oid, A):- underlay(@N,ron_oid,U),
    A:=ron_oid::U.
```

Finally, data plane forwarding rules may also need to be slightly changed. We update the RON forwarding rules `snd` and `rcv` from Section 4.1.2 in the context of layering:

```
snd ron_oid.send(@Next,Dest,Packet) :- ron_oid.send(@N, Dest, Packet),
    ron_oid.RT(@N, Dest, Next),
    underlay(@N, ron_oid, Local),
    Local!=Dest.
rcv ron_oid.recv(@N, Packet) :- ron_oid.send(@N, Dest, Packet),
    underlay(@N, ron_oid, Local),
    Local==Dest.
```

The local address stored in `localAddr` is replaced by `underlay(ron_oid,Local)`, where `Local` is the current underlay address of the overlay `ron_oid`. Note that while the above rules achieve the same functionality as the previous two rules in Section 4.1.2, they are more flexible in allowing packets to route over underlays that can be switched at runtime.

6.1.3 Bridging

Language-level support for bridging is accomplished in either of two ways. The pseudo code is shown in Algorithm 6.3. In the pre-configured method (Line 11, 19), the gateway `GwAddr` for overlay `oid` is stored in the table. MOSAIC routes a packet designated to overlay `oid` towards `Addr`, and the process repeats recursively until the gateway is reached; at that point, the `forward` table will no longer have an entry for the overlay `oid`, and instead it will route the packet according to its own policy. If `GwAddr` is set to a static IP address, this is equivalent to setting up an IP tunnel to the gateway. If `Addr` is an anycast address, e.g. `oasis_oid::oid`, the forwarding plane will invoke the Oasis anycast service to locate the closest `oid` overlay node from the current node, and use it to enter the overlay.

Alternatively, in the on-demand method, a source route can be set up for each packet (Line 7-9,15-17). *Mozlog* supports an address type of the form `sr::[gateway, dest]`, which explicitly describes the data path in terms of logical addresses. All nodes will automatically handle the forwarding of such messages to the next recipient in the path.

For example, node A is hosted in an internal network with an internal IP address `ip_a`. Thus its address is recorded in the netAddress table as `(a_net_id, ip_a)`. Here `a_net_id` is a unique identifier of A's internal network. Recognizing that `ip_a` is an internal IP, the composition server will create a routing path via the gateway node that sits on both the Internet and the internal net to bridge the two networks. The bridged network address is encoded in the source routing format as `sr::[ip_gw, ip_a]` and stored in the `netAddress` table. If we layer RON over the source routing address, node A can immediately join a RON network without a public IP address.

6.1.4 Feature Interaction

As discussed in Section 2.2.5, certain networks may not be compatible to be composed together. As a result, it is desirable to reject incompatible compositions at compile stage. A full-fledged automatic feature interaction analysis is outside the scope of this dissertation, and is considered as part of the future directions in Section 8.2.1. In this dissertation, we propose a simple solution to aid compatibility checking for layering. The idea is to let users to create a knowledge base, in which rules specify that which overlay networks are not allowed to layer on top of which networks. Note that the restriction is *directional*. For instance, when network A over network B is illegal,

Algorithm 6.3 pseudo code for bridging-related rule generation

```
1  function bridging(first, second, gw, bn):
2  input: network first, network second, gateway gw, bridge name bn
3  output: bridging related rules
4  for n in first.nodelist:
5    if n ≠ gw:
6      if config.bridging="on-demand":
7          addRules(n, "netAddress(@n,bn,Addr):-netAddress(@gw,second.oid,GwAddr),
8             netAddress(@n,first.oid,MyAddr),
9             Addr:=sr::[GwAddr,MyAddr]".)
10     else: //pre-configured method
11         addRules(n, "forward(@n,second.oid,Addr):-netAddress(@gw,first.oid,GwAddr).")
12 for n in second.nodelist:
13   if n ≠ gw:
14     if config.bridging="on-demand":
15         addRules(n, "netAddress(@n,bn,Addr):-netAddress(@gw,first.oid,GwAddr),
16            netAddress(@n,second.oid,MyAddr),
17            Addr:=sr::[GwAddr,MyAddr].")
18     else: //pre-configured method
19         addRules(n, "forward(@n,first.oid,Addr):-netAddress(@gw,second.oid,GwAddr).")
```

network B over network A may still be legal. The constraints in the knowledge base are manually constructed, and are based on the implementation of the overlay, rather than (automatically) inferred from overlay attributes or features. Such constraints are stored in a global table `kb(X,Y)`, where attribute `X` and `Y` are overlay names, and `X` is not allowed to be layered on `Y`.

Algorithm 6.4 pseudo code for composition compatibility check for layering

```
input: links from the specification
global: knowledge base kb
output: table conflict, empty when no conflict detected.

l1 layer(A, B) :- links(Type, A, B), Type=="Layer".
l2 layer(A, C) :- layer(A, B), layer(B, C).
l3 conflict(A, B) :- layer(A, B), moduleName(A,X), moduleName(B, Y), kb(X, Y).
```

The pseudo code for composition compatibility check is shown in Algorithm 6.4. We use the deductive *Mozlog*-like syntax because it is straightforward to express the constraints in such a manner. Rule `l1` takes the composition input from the composition specification, finds the layering links, and stores them in the `layer` table. Rule `l2` populates the `layer` table by inserting multiple-level layering relationship recursively in table `layer`. Finally, rule `l3` runs a query on table `layer` and `kb` jointly to find illegal composition relationship, if any, and stores it in the table `conflict`.

overlay id	address
alice_net	alice_internal_ip
br1	sr::[alice_gateway_ip, alice_internal_ip]
br2	sr::[ron::alice_gateway_ip, alice_internal_ip]
i3_oid	i3_oid::alice_id

Table 6.1: netAddress table at Alice

Because variables `A` and `B` are overlay IDs while table `kb` stores incompatible overlay *names* (*e.g.* RON, *i*3, etc), the predicates `moduleName` are used to convert from IDs to overlay names. As a result, a non-empty table `conflict` means the composition specification should be rejected.

6.2 Composition Examples

We now demonstrate MOSAIC's ability to support flexible overlay compositions including bridging, layering and hybrid compositions. We present two examples, one that revisits the mobile VoIP example introduced in Section 1.1, and a second example that illustrates dynamic composition.

6.2.1 VoIP between Alice and Bob

Consider the example mentioned in Section 1.1. An overlay composition can solve the problem. Suppose there is a publicly available *i*3 overlay network, and Alice uses her gateway node at home to form a private RON network with Bob and her other friends. Alice and Bob agree on the composition specification shown in Figure 3.3. Based on the overlay specification, MOSAIC generates the *Mozlog* rules to compose overlays together.

Because Alice's situation mirrors Bob's, we use Alice's rules and network state to explain the composition process. First, at Alice's gateway, we configure the RON overlay network over IP as:

```
c1 underlay(ron_oid,A):-netAddress(0,A).
```

We then use bridging to create publicly reachable addresses `br1` and `br2` as shown in Table 6.1. `br1` bridges the internal network AliceNet with the public IP network, and `br2` bridges AliceNet with the RON network.

Finally, we layer *i*3 over the bridged networks we create. Because Alice wants to have reliability for VoIP, we choose the bridging overlay with `BR2` as *i*3's underlay. The composition rules deployed at the Alice node is as follows:

overlay id	address
0	alice_gateway_ip
alice_net	alice_gw_internal_ip
ron_oid	ron_oid::alice_gateway_ip

Table 6.2: netAddress table at Alice's gateway

```
c2 underlay(i3_oid,A):-netAddress(br2,A).
```

When Bob initiates a VoIP call to Alice, he first uses Alice's *i*3 ID to look up her public trigger, and sends traffic to Alice via *i*3's indirection path. After they have located each other, they switch to the *i*3 shortcut data path as the underlay network specifies, which is layered on top of RON and can traverse internal networks (e.g., those behind NATs) using source routing along the gateways.

6.2.2 Dynamic Composition of Chord over IP and RON

To illustrate dynamic composition, we use the Chord DHT to show the benefit of dynamically switching the underlying data path from IP to RON. In Chord, temporary network failures may create non-transitive connectivity between the nodes, possibly creating problems such as invisible nodes, routing loops and broken return paths [27]. Instead of altering the DHT protocol, an alternative is to layer Chord over a resilient routing protocol such as RON that eliminates non-transitivity. Layering Chord over RON can be viewed as trading scalability for performance.

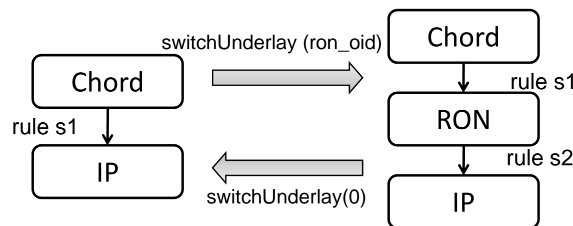

Figure 6.1: Dynamic composition of Chord over two different underlays (IP and RON).

The following rules define two type of layering: Chord over IP and Chord over RON (See Figure 6.1 for the graphical illustration):

```
s1 underlay(chord_oid,A):- netAddress(OID,A),
    switchUnderlay(OID).
s2 underlay(ron_oid,A):- netAddress(0,A).
```

In `s1-s2`, we added a `switchUnderlay(OID)` predicate to switch Chord's underlay to that indicated by the `OID` variable. This `switchUnderlay` can itself be triggered by an event sent from the administrator based on changes to the overlay specifications. Rule `s1` indicates that Chord uses IP as the underlying address when `OID` is 0, and RON when `OID` is `ron_oid`. Rule `s2` defaults RON to use IP at all times. To switch between the two layering schemes, one only needs to generate `switchUnderlay` accordingly.

Dynamic switching is useful because the trade-off between scalability and performance is at the discretion of the Chord administrators, who can make decisions based on network conditions, requirements, etc. Suppose a new overlay providing both resiliency and scalability (*e.g.* SOSR [32]) is available later, one can switch Chord's underlay from RON to the new one to further improve scalability. Unlike restarting Chord from scratch, dynamic switching preserves existing state in the network such as key/value pairs without disrupting the DHT lookup service. Once the Chord underlay network address is changed on a node, the stabilization process will propagate it to the node's successors, predecessor and other nodes that have it in its finger table. We present our experimental evaluation of this example in Section 6.3.2.

6.2.3 Multicast for Pub/Sub Services

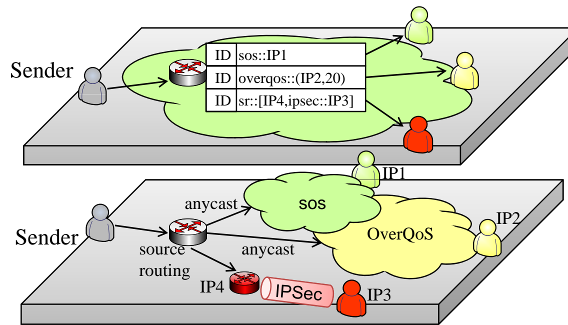

Figure 6.2: A publish/subscribe composition example using multicast.

Consider a stock broker that publishes a stream of stock prices, and sends to the subscribers via multicast. Some of the subscribers are active traders, who demand the data in a timely fashion. Some of the users are under DoS attack. Some of them are connected via unsecure wireless links. In Mosaic, we can compose the existing *i*3, SOS, OverQoS and IPSec overlays, as in Figure 6.2.

At the top layer, the publisher and the subscribers join an *i*3 overlay. The published stock

price has a unique *i*3 ID, which is known to the subscribers. The subscribers form a multicast tree in *i*3 and insert overlay-specific addresses into their leaf triggers. The traders use `overqos::(IP2, 20ms)` to attain quality of service guarantees (latency within 20ms); the users under DoS attack use `sos::IP1` to allow packets with stock prices pass through, while blocking other unwanted IP traffic. Users with unsecure connections may redirect the traffic through an IPSec tunnel by inserting `sr::[IP4, ipsec::IP3]` into the trigger. In this scenario, the data plane of *i*3 is layered over different overlays as well as IP; however, *i*3 is unaware of this.

6.3 Evaluation

In this section, we present the evaluation of MOSAIC on a local cluster and on PlanetLab. First, we validate that *Mozlog* specifications for declarative networks, compositions, tunneling and packet forwarding are comparable in performance to native implementations. Second, we use our implementation to demonstrate feasibility and functionality, using actual legacy applications that run unmodified on various composed overlays using MOSAIC.

In all our experiments, we make use of a declarative Chord implementation which consists of 35 rules. Our *i*3 implementation uses Chord and adds 16 further rules. We also implement the RON overlay in 11 rules. Both *i*3 and RON can be used by legacy applications via the `tun` device, as described in Section 4.1.4.

6.3.1 Wide-area Composition Evaluation

We deployed MOSAIC on PlanetLab to understand the wide-area performance effects of using the system. We purposely chose a composed overlay including *i*3, RON, source routing, and tunneling for legacy applications (all implemented within MOSAIC in 69 *Mozlog* rules) to bring the Alice example from the introduction and Section 6.2.1 to a resolution.

Our experimental setup is as follows. As our end-host, we used a Linux PC in Edison, NJ with a high speed cable modem connection as the gateway node, which performed NAT for a Thinkpad X31 laptop. The laptop functioned as our server, using Apache to serve a 21MB file. The file was downloaded from Salt Lake City, UT with a modified version of `wget` that records the download throughput.

These two nodes in NJ and UT, plus three additional nodes in Philadelphia, Berkeley, and

Ithaca, were used to form a private RON network. We further selected 44 nodes from PlanetLab, mostly in the US, to run *i*3. During the experiment, in order to validate the functionality of resilient routing provided by RON, we manually injected network failures by changing the firewall rules on the gateway to block the downloader's traffic 30 seconds after `wget` was started; then we unblocked the traffic after another 30 seconds. For the purposes of comparison with the best case scenario, we repeated the same test using direct IP communication. Note that direct IP loses all the benefits of our composed overlay (no resilience, NAT, or mobility support), but achieves the best possible performance. Since our server was behind a NAT, in the direct IP experiment, we had to manually set up a TCP port forwarding rule on the gateway node to reach the Apache server. We repeated multiple runs of the experiments and observed no significant differences.

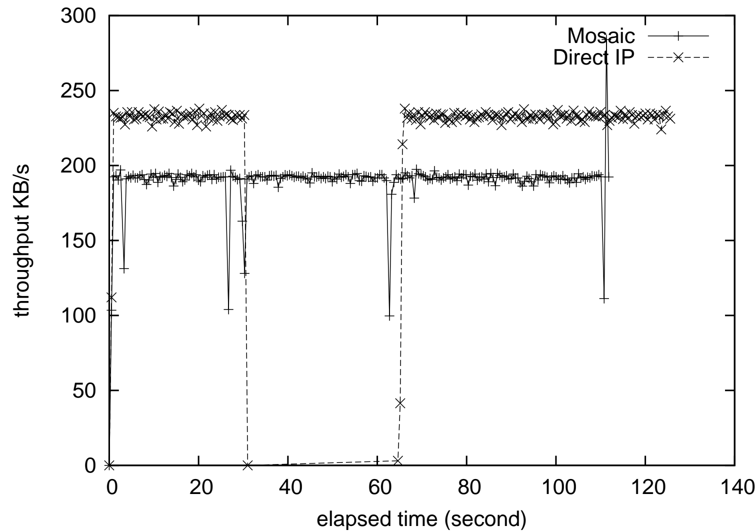

Figure 6.3: Throughput comparison between overlay composition in Mosaic vs direct IP connection during network failure. Network failures were injected 30 seconds after experiment start, and removed after 30 additional seconds.

Figure 6.3 shows the throughput of the download over time for Mosaic and DirectIP. We make the following observations. First, Mosaic's performance over the wide area is respectable: Despite implementing the *entire* composed overlay (including legacy support for applications using Mosaic) in *Mozlog*, we incurred only 20% additional overhead compared to using direct IP, while achieving the benefits of mobility, NAT support and resilient routing. The majority of the overhead comes from the extra packet headers for the composed overlay protocols—an overhead that is repaid with significant functionality. Second, with respect to the functionality of our composed overlay,

we were able to achieve successful downloads from a server behind a NAT using Mosaic. In addition, resilient routing was achieved: Our RON network periodically monitored the link status and recovered from routing failures. Hence, during the period where we injected the routing failures, Mosaic was able to make a quick recovery from failure, as is shown by the sustained throughput. On the other hand, DirectIP suffered a failure (and hence a drop of throughput to zero) during the 30-60 second period. Overall, Mosaic was able to complete the download in a shorter time despite lower throughput, due to the resiliency of RON.

6.3.2 Dynamic Overlay Composition

In our final experiment, we evaluate the dynamic composition capabilities of Mosaic. Our setup consists of an 8-node cluster, where each node has a similar hardware configuration to the setup in Section 5.6.1.

As a baseline prior to the dynamic switching experiment, we made static comparisons between two composed networks: we executed *Chord-over-IP* and *Chord-over-RON* on our cluster, which consists of the Chord overlay on top of IP and RON respectively. Our network size is 16, where each machine executed two instances of the composed overlay nodes. In the steady state, each node periodically issues a lookup request. A lookup is *accurate* if the results of the lookup are correct, *i.e.*, the results point to the node whose key is the closest successor of the lookup key. Based on this definition, we compute the lookup accuracy rate, which is the fraction of accurate lookups over the duration of each experimental run at every 1 minute interval. Network link failures are emulated by changing the firewall settings in the cluster to drop packets between the selected nodes.

Figure 6.4 shows our evaluation results over a period of 20 minutes, with the first link failure at the 7th minute, then the second link failure at the 10th minute, and the failures recovered at the 16th minute. When the first link failure occurred, we observed that lookup accuracy of *Chord-over-IP* dropped to 93%. The accuracy further dropped to 86% when the second link failure occurred, only to recover when network connectivity was reestablished. On the other hand, *Chord-over-RON* continued to sustain high lookup accuracy ($> 99\%$) even in the face of network failures, due to its ability to find alternative routes quickly.

Having compared the composed overlays separately, we next evaluate Mosaic's dynamic switching capability, where we started with *Chord-over-IP*, and then switched our composition to *Chord-over-RON* after 7 minutes. This dynamic switching is achieved by merely changing the underlay

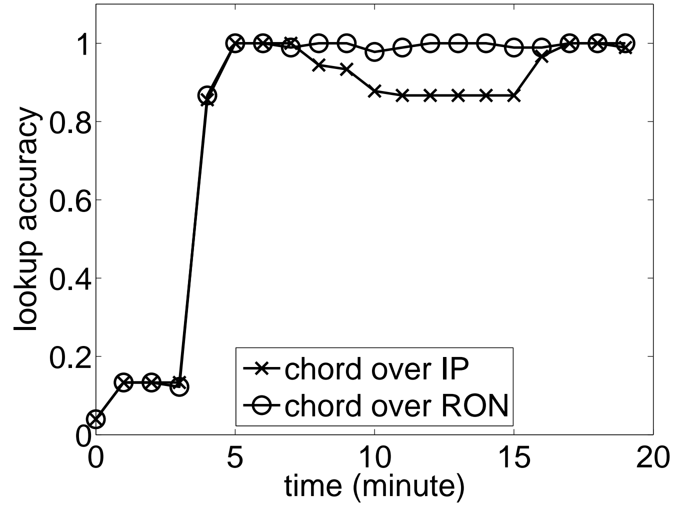

Figure 6.4: Lookup accuracy comparison between Chord over IP and Chord over RON.

address of Chord from IP to RON, as described in Section 6.2. Figure 6.5 shows the resulting lookup accuracy over a period of 15 minutes. We observe that during the process of switching its underlay from IP to RON, Chord continued to sustain high lookup accuracy, demonstrating that Mosaic is able to performing dynamic switching seamlessly.

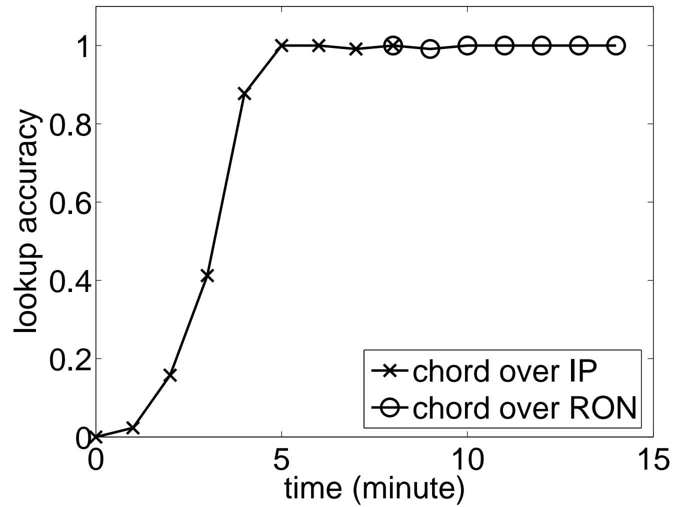

Figure 6.5: Chord lookup performance during dynamic underlay network switching from IP to RON.

6.4 Summary

In this chapter, we first described how to compile high-level network composition specifications into *Mozlog* glue code, and then demonstrated it using several concrete examples. We evaluated the effectiveness of the MOSAIC system by using two real-world experiments, including a wide-area network composition of $i3$, RON to provide a resilient, mobility-aware composed overlay, and dynamic switching between a Chord lookup overlay service over regular IP and Chord over a RON network. The results showed that the performance overhead of MOSAIC is respectable compared to native implementations, while achieving the benefits of network composition.

Chapter 7

Related Work

In this chapter, we compare MOSAIC with other approaches that provide flexible network service introduction.

7.1 Extensible Protocols and Networks

Ritchie's STREAMS system [76] provided an extensible architecture for constructing protocols. It was later shipped with the AT&T System V version of UNIX, and was generalized into stackable protocol architectures for streams of data. Code adhering to a message handling discipline shared by all such STREAMS modules could be pushed onto, and subsequently popped from, a logical stack of processing modules through which streams of message data would pass.

The x-kernel [38] is a framework for implementing and composing network protocols. It provides a collection of protocol elements (called micro-protocols) along with a generic mechanism for composing them based on layering. It demonstrated that large protocols could be represented in the form of many smaller components with the layering model while still achieving high performance. It is intended to be used at end nodes, where packet motion is vertical (between the network layers and user level) rather than horizontal (between network hosts).

Protocol boosters [21] also provide flexible network processing within the network by altering protocol configurations. Protocols are modeled in layers in protocol boosters and sub-layers are allowed to be inserted between two points so that it is transparent to the remainder of the protocol stack. By adaptively inserting or deleting sub-layers based on local conditions, network performance

may be improved. For example, audio or video transmission over a wireless link with high loss rate may be "boosted" by a forward error correction (FEC) module.

PIP [24] exploits the flexibility of source routing as a network evolution mechanism, which requires an expanded packet header and a different forwarding mechanism that tracks the active destination in the list of source routes. PIP was proposed as a candidate during the design of IPv6. It can also be used to support a variety of routing services, including mobility, and provider selection. The bridging mechanism in MOSAIC resembles PIP's source routing scheme in many ways.

Active networks [87, 83] allow anyone to write code that will affect a router. Untrusted mobile code is usually embedded in the packets as in the capsule programming model [92] and executed on the active routers along the routing path. Since then, new technologies had arisen, in particular new programming language [63, 29] and security technologies [12] that could provide desirable sets of tradeoffs amongst security, programmability, usability and performance. Some of the representative projects include secure active extension environment (ALIEN [6] and SANE [5]), packet language for active networks (PLAN [34], SNAP [64]), etc.

Compared with MOSAIC, active networks are more flexible in terms of dynamic service introduction but also raises many concerns in security because of the untrusted code. In MOSAIC, only trusted users can deploy code in the infrastructure. Besides, the code are not arbitrary code but database style queries, running inside a query engine sandbox to ensure safety.

The effort on peer-to-peer style systems mostly in data and file sharing services promoted the idea of introducing new services using an overlay. Peers use the IP network layer as a virtual link; each peer serves as a router in the overlay. Arbitrary distributed computing architectures can be built because these peers are fully programmable nodes without restriction. Early systems such as X-Bone [88] and the most recent and popular testbed PlanetLab [70] have stimulated many overlay network designs that go beyond file sharing. Examples of new capabilities include DDoS resistance [44, 81], performance and reliability [80, 7], and QoS [86]. Overlays such as Internet Indirection Infrastructure ($i3$) [84] and TRIAD [30] provide a deployable solution for disruptive data forwarding capabilities. A list of features provided by overlay services can be found in Section 2.2.1.

Aside from what new capabilities overlays can provide, researchers have also studied how to design new environments to simplify the development and deployment of new overlay networks. MACE [45] (formerly known as MACEDON [77]) is a domain-specific C++ language extension to

describe a distributed system's behavior from which real operating code can be generated. It also provides a model checker capable of finding violations of liveness properties which lead the system to dead states [46].

Declarative networking [56] took a different approach to design networks from a database perspective. The entire network state is modeled as structured data from a distributed database. In this setting, network protocols are equivalent to querying and manipulating relevant state in a distributed manner. Therefore, the implementation of a network protocol can be viewed as the result of a distributed database query processing. In declarative networks, a domain-specific programming language *NDlog* based on deductive database query languages is proposed to specify network protocols. *NDlog* programs are demonstrated to be an order of magnitude shorter than the counterpart imperative implementations. The language itself is even powerful enough (with minor extensions) to serve as a meta-compiler to compile *NDlog* programs [15], and work in emerging network environments such as mobile adhoc networks (MANET) [52] and sensor networks [14]. In particular, the P2 system [55] is the implementation of the declarative engine for overlay networks. It takes an *NDlog* program and compile them into dataflow runtime graphs which in principle ensemble the Click extensible router dataflows [47].

The foundation of the dissertation is built on the declarative networking concept. Compared with P2, Mosaic not only uses the declarative language to do quick prototyping of new overlay protocols, but also achieves interoperability among existing overlays by using simple yet flexible query-style interfaces between networks. This allows the system to provide automation in composing different networks together to implement a combination of multiple features. The *Mozlog* language provides new features to support composition, as well as legacy application support. Finally, the Mosaic runtime system performance is optimized by both novel compiler-based techniques and careful engineering efforts so that query executions are up to 10 times faster than P2. Interestingly but not surprisingly, similar data accessing and optimization techniques exist in distributed database and its query optimization [48]. For example, accessing data in CView can be viewed as query with limited access patterns [23]; the inline view expansion optimization in Section 4.2.3 resembles "view unfolding" in database; the binding patterns and source capability descriptors have also been studied by Levy *et al.* [50].

7.2 Network and Service Composition

Composing a plurality of heterogeneous networks was proposed in Metanet [93], and also examined in Plutarch [17]. One of the implementation examples to connect multiple networks together is AVES [67]. Oasis [59] and OCALA [42] provide legacy support for multiple overlays. Oasis picks the best single overlay for performance. OCALA proposes a mechanism to *stitch* (similar to MOSAIC's bridge functionality) multiple overlay networks at designated gateway nodes to leverage functionalities from different overlays. There are also many projects that focus on application level service composition, with different emphasis, such as adaptive configuration based on network conditions [28], fault tolerance and personalization [61], performance and QoS awareness in P2P environment [31].

In contrast, MOSAIC's primary focus is on overlay network specification and composition within a single framework. As a result, MOSAIC is complementary to OCALA and Oasis. MOSAIC's use of a declarative language results in more concise overlay network specification and composition, whose performance is quite comparable to native code. MOSAIC also provides support for layering in addition to bridging. Finally, MOSAIC is not limited to IP-based networks, supports dynamic composition, and routing primitives such as unicast and multicast. These benefits result in better extensibility and evolvability of MOSAIC over existing composition systems.

Another class of composition work is *Web service composition*, as surveyed by Milanovic and Malek [62]. Each Web service serves like a remote procedure call over HTTP or HTTPS, and provides a relatively basic functionality, which is described with additional pieces of information, either by a semantic annotation of what it does and/or by a functional annotation of how it behaves. The industry standard specification is Web Service Definition Language (WSDL) [13]. A complicated application logic is built by composing multiple Web services together, which is described separately in a flow specification, such as Business Process Execution Language for Web Services (BPEL4WS) [8]. The process of obtaining the composition flow is either manual or in some cases can be assisted by an AI planning software in the context of semantic Web.

Compared with network service composition, the major difference is that Web services are best described with request/response models, where usually only a single entity is involved in utilizing a service (The service may still be provided by multiple providers.) On the other hand, network services are based on send/receive models, where two (a sender and a receiver) or more (in the case

of multicast/broadcast) entities participate the process. However, the goal based AI planning work and automatic composition in Web services may provide a viable path towards automatic network composition.

Chapter 8

Conclusions

8.1 Conclusions

In this dissertation, we presented Mosaic, an extensible infrastructure that enables not only the specification of new network services in a data-centric, declarative language, but also dynamic selection and composition of such networks. Mosaic provides *declarative networking* as a fixed model of computation rather than a fixed service: it uses a unified declarative language (*Mozlog*) and runtime system to enable specification of new overlay networks, as well as their composition in both the control and data planes. It enables the use of many special-purpose services and their compositions rather than one general-purpose, monolithic architecture that serves all purpose. We demonstrated Mosaic's composition capabilities via deployment and measurement on both a local cluster, and the Emulab and PlanetLab testbed, and showed that the performance overhead of Mosaic is respectable compared to native implementations, while achieving the benefits of network composition.

Specifically, the dissertation makes the following contributions:

- A unifying network architecture (Mosaic) under which new networks can be developed, deployed, selected, and dynamically composed according application and administrator needs.

- A declarative programming language (*Mozlog*) to concisely specify high-level network protocol specifications.

- A runtime system prototype that can translate *Mozlog* specifications into efficient implementations.

8.2 Open Questions and Future Directions

In this section, we discuss some open questions and future directions in order to make network architectures like MOSAIC into reality.

8.2.1 Network Feature Interaction

Different network protocols provide different services, with different approaches. Two networks are *incompatible* if they provide conflicting goals or use conflicting approaches. To compose incompatible networks together may negate the effectiveness of the individual services. For example, if network A provides a routing service with minimal latency, while network B provides a routing service with maximum bandwidth, layering them together as a composed network may deliver neither feature. Consider another example where two end-to-end services use encryption and compression respectively. If a packet is encrypted before it is compressed, the compression algorithm can hardly achieve any benefit, while doing compression before encryption is perfectly fine. That is, the order of the service composition may also lead to undesirable feature interaction.

Therefore, an ideal composable network architecture should be able to validate the compatibility of a proposed network composition and guide the users towards effective composition. In this dissertation, we took a first step. We assume some experts to study the interactions for all possible composition scenarios in layering, and store the results in a knowledge database for users to query. Clearly such an approach is not scalable when the number of composable network elements grows. It also does not analyze the interactions between networks that are bridged together. Analyzing bridged network interaction is sometimes harder than analyzing layered network interaction because the bridged networks usually cover different physical networks with different properties in terms of latency, throughput, confidentiality, etc. These factors make the analysis more complicated. An advanced approach is to automatically infer the features of those network elements, and deduce the interaction compatibility between the features. It is known to be very hard to analyze programs written in imperative low-level language like C. On the other hand, writing a composable network service in a high-level declarative language like *Mozlog* provides the opportunity for such

feature analysis. Previous work on feature interaction in the areas of security analysis [18] and telecommunications services [40, 22] may be inspiring.

Finally, another potential future direction is to provide automatic composition for given features, given application requirements, network properties and constraints. It is desirable when selecting and configuring composable networks from a large pool of potential services is beyond what a user can handle. When network condition changes, the composition should dynamically change accordingly on the fly to adapt to the new situation.

8.2.2 Composition

In this dissertation, we mainly focused on programming and composing network layer services. In general, there are more composition opportunities to explore. First, there are many overlay network protocols that work on the transport or session layers [60, 51, 95]. The challenges are to define appropriate interfaces and let them interact well with those network layer overlays in composition. Second, a network protocol usually consists of several smaller components, such as routing, congestion control aggregation, state storage and buffer management [16]. Composing in terms of network protocol components provides finer granularity and more flexibility, but at the potential expense of further complicated feature interaction analysis. Finally, dynamic network composition scenarios deserve more investigation. In this dissertation, we proposed to impose a constraint on dynamic composition in Section 3.3.3: all logical-to-physical node mapping before and after underlay switching should remain the same. In general, the effect of dynamic switching is overlay specific. If an overlay already has robust state management algorithm built in, it will probably not be affected by node mapping changes. More study cases need to be examined and it is desirable to come up with necessary and sufficient conditions for correct dynamic composition.

8.2.3 Performance

In performance evaluation, we noticed that Mosaic runtime system achieves respectable performance in wide-area network systems, but still impose relatively high CPU overhead in local area networks with gigabit connectivity. We have discussed a few optimization techniques in Section 4.2.3. There are a few other similar techniques such as loop unrolling, when applying to dataflow element loops, may also be implemented. While more compiler optimization techniques might further

improve the performance, the intrinsic processing overhead in any dynamic software router is inevitable. An interesting direction is to optimize via specialized hardware platforms. By augmenting the *Mozlog* compiler, one might be able to run the same declarative protocol specification using special high performance network processors [89] to boost the performance by several orders of magnitude, without learning the complicated hardware specific programming, *e.g.* distinguishing the fast path and slow path. On the other hand, it is also interesting to study the right hardware acceleration mechanisms that can specifically optimize a data-centric query programming language for network protocols.

8.2.4 Programming Paradigm

The Mosaic architecture is based on a domain specific programming language in the declarative logic-style programing paradigm. While the paradigm has many advantages as discussed in the dissertation, it does have a steep learning curve to typical network protocol designers. The open question is: what is the right programming paradigm(s) for composing network services? After all, many research projects also showed advantages in imperative programming paradigm [45] and functional programming paradigm [9, 66, 34]. In fact, the special predicates provided in *Mozlog* resemble imperative system calls in many ways, and most of the database operators can also be rewritten in the compositions of several higher-order functions, such as `map` for assignments, `reduce` for aggregations, and `filter` for conditions, etc. In the future, the language for network programming may well support multiple programming paradigms. It should also be intuitive for human designers to program, and feasible for computers to analyze as described in Section 8.2.1.

Appendix A

Mozlog Grammar and Syntax

The grammar that specifies the *Mozlog* language syntax is described in Figure A.1, Figure A.2, and Figure A.3.

The following conventions are used when describing the grammar:

- Lower-lettered phrases in `typewriter font` are grammar classes.
- Lower-lettered phrases in *italic font* are literal constants.
- Precedence is lower for alternative forms that appear later.
- Upper-lettered phrases are literal constants generated by the lexer. In particular, `NAME` represents the name of a predicate beginning with a lower-case letter; `AGG` represents the name of an aggregation operator (`min`, `max`, `count`, `sum`, `avg` are currently supported.); `VALUE` represents numerical constants; `STRING` represents string constants in double quotes; `VAR` represents variable names starting with a upper-case letter. It may also be the symbol "_" which read as do-not-care; `FUNC` represents function names beginning with "`f_`".

```
program     ::=  clauselist

clauselist  ::=  clause
               | clause clauselist

clause      ::=  rule
               | fact
               | materialize
               | watch
               | cview

materialize ::=  materialize(functorname,tablearg,tablearg,primkeys).

tablearg    ::=  VALUE

primkeys    ::=  keys(keylist)

keylist     ::=  VALUE
               | VALUE, keylist

watch       ::=  watch(NAME).

cview       ::=  def functor { clauselist }

fact        ::=  functor.

rule        ::=  NAME functor :- termlist .
               | NAME delete functor :- termlist.

termlist    ::=  term
               | term, termlist

term        ::=  functor
               | assign
               | select
```

Figure A.1: The grammar: program declarations

```
functor     ::=  functorname functorbody
               | functorname@atom functorbody

functorname ::=  VAR
               | this.VAR

functorbody ::=  (functorargs)

functorargs ::=  functorarg
               | functorarg, functorargs
               | @ atom
               | @ atom, functorargs

functorarg  ::=  atom
               | aggregate
               | & atom

aggregate   ::=  AGG<VAR>
               | AGG<@VAR>
               | AGG<*>

function    ::=  FUNC(funcargs)
               | FUNC()

funcargs    ::=  funcarg
               | funcarg, functionargs

funcarg     ::=  math_expr
               | atom

select      ::=  expr

assign      ::=  VAR := expr
```

Figure A.2: The grammar: functor declarations

```
expr        ::=  atom
               | function
               | ( expr )
               | unaryop expr
               | expr binaryop expr
               | [ ]
               | [ exprlist ]

exprlist    ::=  expr
               | expr,exprlist

unaryop     ::= not

binaryop    ::=  + | - | * | / | and | or | ::
               | == | != | > | <   | >=  | <=

atom        ::= VALUE | VAR | STRING | NULL
```

Figure A.3: The grammar: expressions

Appendix B

Composition Specification

B.1 Composition Specification Example

The following XML code is a demonstration of the composition specification from the Alice-and-Bob example.


```
<mosaic>
 <bindings>
  <subnet>
   <name>AliceSubNet</name>
   <ip>10.1.1.0</ip><mask>255.255.255.0</mask>
  </subnet>
  <subnet>
   <name>BobSubNet</name>
   <ip>10.2.1.0</ip><mask>255.255.255.0</mask>
  </subnet>
  <node><name>AliceGW</name>
    <ip id="AliceSubNet">10.1.1.1</ip>
    <ip id="0">123.45.67.8</ip>
  </node>
  <node> <name>AlicePC</name>
    <ip id="AliceSubNet">10.1.1.12</ip>
  </node>
  <node><name>BobGW</name>
    <ip id="0">234.56.78.1</ip>
  </node>
 </bindings>
 <composition>
  <module id="AliceNet" Name="IP" type="New">
    <constraints>
     <subnet>AliceSubnet</subnet>
    </constraints>
    <nodelist>
      <node>AliceGW</node>
      <node>AlicePC</node>
    </nodelist>
    <gateway>AliceGW</gateway>
  </module>
  <module id="BobNet" Name="IP" type="New">
```

```
    <constraints>
     <subnet>BobSubnet</subnet>
    </constraints>
    <nodelist>
      <node>BobGW</node>
      <node>BobPC</node>
    </nodelist>
    <gateway>BobGW</gateway>
  </module>
  <module id="ron_oid" name="RON" type="Extend">
    <nodelist>
     <node>AliceGW</node> <node>BobGW</node>
    </nodelist>
    <attributes>
     <resiliency>redundancy</resiliency>
    </attributes>
    <code ref="http://www.mosaic-system.net/ron/v1"/>
  </module>
  <module id="i3_oid" name="i3" type="Extend">
    <nodelist>
     <include>AliceGW</include> <include>BobGW</include>
    </nodelist>
    <attributes>
      <mobility>nearestClientProxy</mobility>
    </attributes>
    <code ref="http://www.mosaic-system.net/i3/v1"/>
    <gateway></gateway>
  </module>

  <link type="bridge" name="AliceBridge">
    <module>AliceNet</module>
    <module>RON</module>
    <gateway>AliceGW</gateway>
  </link>
  <link type="bridge" name="BobBridge">
    <module>BobNet</module>
    <module>RON</module>
    <gateway>BobGW</gateway>
  </link>
  <link type="layer" topmodule="i3_oid" bottommodule="ron_oid" />
  <link type="layer" topmodule="ron_oid" bottommodule=0 />
 </composition>
</mosaic>
```

B.2 Abstract Syntax of Composition Specification

Figure B.1 illustrates the abstract syntax of the composition specification.

```
spec        ::=  Spec(bindings,composition)

bindings    ::=  binding
               | binding, bindings

binding     ::=  subnet | node

subnet      ::=  Subnet(NAME,ADDR,MASK)

node        ::=  Node(NAME, addrlist)

addrlist    ::=  addr
               | addr, addrlist

composition ::=  Composition( modules, links )

modules     ::=  module | module, modules

module      ::=  existing | new | extend

existing    ::=  Existing(NAME, OID)

new         ::=  New(TYPE, NAME, [nodelist] <, Gateway=NAME, URL>)

extend      ::=  Extend(NAME, OID, [nodelist])

nodelist    ::=  NAME | NAME, nodelist

links       ::=  link | link, links

link        ::=  layering | bridging

layering    ::=  Layering NAME Over NAME

bridging    ::=  Bridging NAME Over NAME < Via NAME> As NAME
```

Figure B.1: Abstract syntax of composition specification

Bibliography

[1] Martin Abadi and Leslie Lamport. Composing Specifications. In *Stepwise Refinement of Distributed Systems - Models, Formalisms, Correctness*, volume 430, pages 1–41. Springer-Verlag, 1989.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, pages 186–201, 1999.

[4] Akamai Inc. The Akamai content distribution network. http://www.akamai.com.

[5] D. S. Alexander, W. A. Arbaugh, A. D. Keromytis, and J. M. Smith. A Secure Active Network Environment Architecture: Realization in SwitchWare. *IEEE Network Magazine, special issue on Active and Programmable Networks*, 12(3):37–45, 1998.

[6] D. Scott Alexander, Marianne Shaw, Scott M. Nettles, and Jonathan M. Smith. Active Bridging. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, 1997.

[7] David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris. Resilient Overlay Networks. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[8] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. Business Process Execution Language for Web Services. 2003. http://www.ibm.com/developerworks/library/specification/ws-bpel/.

[9] Joe Armstrong, Robert Virding, Claes Wikstrom, and Mike. Williams. *Concurrent Programming in Erlang*. Prentice Hall, second edition, 1996.

[10] Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Looking Up Data in P2P Systems. *Communications of the ACM, Vol. 46, No. 2*, February 2003.

[11] Hari Balakrishnan, Karthik Lakshminarayanan, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Michael Walfish. A Layered Naming Architecture for the Internet. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, 2004.

[12] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *IEEE Symposium on Security and Privacy*, pages 164–173, Oakland, CA, 1996.

[13] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. 2001. http://www.w3.org/TR/wsdl.

[14] David Chu, Lucian Popa, Arsalan Tavakoli, Joseph Hellerstein, Philip Levis, Scott Shenker, and Ion Stoica. The design and implementation of a declarative sensor network system. In *Proceedings of The 5th ACM Conference on Embedded networked Sensor Systems (SenSys'07)*, Sydney, Australia, November 2007.

[15] Tyson Condie, David Chu, Joseph M. Hellerstein, and Petros Maniatis. Evita Raced: Metacompilation for Declarative Networks. In *Proceedings of Conference on Very Large Data Bases (VLDB)*, 2008.

[16] Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Sean Rhea, and Timothy Roscoe. Finally, a Use for Componentized Transport Protocols. In *Proceedings of the Fourth Workshop on Hot Topics in Networks (HotNets-IV)*, College Park, MD, November 2005.

[17] J. Crowcroft, S. Hand, R. Mortier, T. Roscoe, and A. Warfield. Plutarch: An Argument for Network Pluralism. In *Proceedings of ACM SIGCOMM Workshop on Future Directions in Network Architecture (FDNA)*, 2003.

[18] A. Datta, A. Derek, J. C. Mitchell, and D. Pavlovic. A derivation system and compositional logic for security protocols. *Journal of Computer Security (Special Issue of Selected Papers from CSFW-16)*, 13:423–482, 2005.

[19] John R. Douceur and Jon Howell. Distributed directory service in the farsite file system. In *Proceedings of the 7th symposium on Operating Systems Design and Implementation (OSDI'06)*, pages 321–334, Seattle, Washington, 2006.

[20] Emulab. Network Emulation Testbed. 2006. http://www.emulab.net.

[21] D. C. Feldmeier, A. J McAuley, J. M. Smith, D. S. Bakin, W. S. Marcus, and T. M. Raleigh. Protocol Boosters. *IEEE Journal on Selected Areas in Communications (Special Issue on Protocol Architectures for 21st Century Applications*, 16(3):437–444, April 1998.

[22] Amy P. Felty and Kedar S. Namjoshi. Feature specification and automated conflict detection. In *Proceedings of the 6th Feature Interactions Workshop (FIW'00)*, Glasgow, Scotland, 2000. IOS Press.

[23] Daniela Florescu, Alon Y. Levy, Ioana Manolescu, and Dan Suciu. Query optimization in the presence of limited access patterns. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 311–322, 1999.

[24] Paul Francis. A near-term architecture for deploying pip. *IEEE Network*, 7(3):30–37, May 1993.

[25] M. Freedman, K. Lakshminarayanan, and D. Mazieres. OASIS: Anycast for any service. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI'06)*, 2006.

[26] Michael J. Freedman, Eric Freudenthal, and David Mazires. Democratizing Content Publication with Coral. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI'04)*, San Francisco, CA, March 2004.

[27] Michael J. Freedman, Karthik Lakshminarayanan, Sean Rhea, and Ion Stoica. Non-Transitive Connectivity and DHTs. In *Proc. of the Second Workshop on Real, Large Distributed Systems (WORLD'05)*, 2005.

[28] Xiaodong Fu, Weisong Shi, Anatoly Akkerman, and Vijay Karamcheti. Cans: Composable, adaptive network services infrastructure. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS'01)*, 2001.

[29] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison Wesley, 1996.

[30] M. Gritter and D. Cheriton. An Architecture for Content Routing Support in the Internet. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS'01)*, 2001.

[31] Xiaohui Gu, Klara Nahrstedt, and Bin Yu. Spidernet: An integrated peer-to-peer service composition framework. In *Proceedings of the 13th International Symposium on High-Performance Distributed Computing (HPDC-13)*, pages 110–119, Honolulu, Hawaii, June 2004.

[32] Krishna P. Gummadi, Harsha Madhyastha, Steven D. Gribble, Henry M. Levy, , and David J. Wetherall. Improving the reliability of internet paths with one-hop source routing. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI'04)*, 2004.

[33] Alon Y. Halevy, Zachary G. Ives, Dan Suciu, and Igor Tatarinov. Schema mediation in peer data management systems. In *Proceedings of IEEE Conference on Data Engineering (ICDE)*, 2003.

[34] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A Packet Language for Active Networks. In *Proceedings of International Conference on Functional Programming (ICFP)*, 1998.

[35] Yang hua Chu, Sanjay G. Rao, Srinivasan Seshan, and Hui Zhang. A case for end system multicast. *IEEE Journal on Selected Areas in Communication (JSAC), Special Issue on Networking Support for Multicast*, 20(8), 2002.

[36] Yan Huang, Tom Z. J. Fu, Dah-Ming Chiu, John C. S. Lui, and Cheng Huang. Challenges, design and analysis of a large-scale p2p-vod system. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, Seattle, WA, August 2008.

[37] Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham, Boon Thau Loo, Scott Shenker, and Ion Stoica. Querying the Internet with PIER. In *Proceedings of Conference on Very Large Data Bases (VLDB)*, 2003.

[38] N. C. Hutchinson and L. L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, 1991.

[39] John Ioannidis, Dan Duchamp, and Gerald Q. Maguire, Jr. IP-based Protocols for Mobile Internetworking. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, September 4-6 1991.

[40] Michael Jackson and Pamela Zave. Distributed feature composition: A virtual architecture for telecommunications services. *IEEE Transactions on Software Engineering*, XXIV(10):831–847, 1998.

[41] John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and James W. O'Toole, Jr. Overcast: Reliable multicasting with an overlay network. In *Proceedings of the 4th symposium on Operating Systems Design and Implementation (OSDI'00)*, pages 197–212, October 2000.

[42] Dilip Joseph, Jayanthkumar Kannan, Ayumu Kubota, Karthik Lakshminarayanan, Ion Stoica, and Klaus Wehrle. OCALA: An architecture for supporting legacy applications over overlays. In *Proc. NSDI*, 2006.

[43] Dina Katabi and John Wroclawski. A framework for scalable global IP-anycast (GIA). In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, 2000.

[44] A. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure overlay services. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, 2002.

[45] Charles Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin Vahdat. Mace: Language Support for Building Distributed Systems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Diego, CA, June 2007.

[46] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, Death, and the Critical Transition: Detecting Liveness Bugs in Systems Code. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI'07)*, Cambridge, MA, April 2007.

[47] 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, 2000.

[48] Donald Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, 2000.

[49] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones. SOCKS Protocol Version 5. RFC1928, 1996.

[50] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proceedings of the Twenty-second International Conference on Very Large Databases*, pages 251–262, Bombay, India, 1996.

[51] Yi Li, Yin Zhang, Lili Qiu, and Simon S. Lam. SmartTunnel: Achieving reliability in the internet. In *Proceedings of Annual Joint Conference of the IEEE Computer Communications Societies (INFOCOM)*, 2007.

[52] Changbin Liu, Yun Mao, Mihai Oprea, Prithwish Basu, , and Boon Thau Loo. A declarative perspective on adaptive manet routing. In *Proceedings of ACM SIGCOMM Workshop on Programmable Routers for Extensible Services of TOmorrow (PRESTO)*, Seattle, WA, August 2008.

[53] Boon Thau Loo. The Design and Implementation of Declarative Networks (Ph.D. Dissertation). Technical Report UCB/EECS-2006-177, University of California at Berkeley, 2006.

[54] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative Networking: Language, Execution and Optimization. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, June 2006.

[55] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing Declarative Overlays. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[56] Boon Thau Loo, Joseph M. Hellerstein, Ion Stoica, and Raghu Ramakrishnan. Declarative Routing: Extensible Routing with Declarative Queries. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, Philadelphia, PA, 2005.

[57] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Design of an acquisitional query processor for sensor networks. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 491–502, 2003.

[58] Harsha V. Madhyastha, Tomas Isdal, Michael Piatek, Colin Dixon, Thomas Anderson, Arvind Krishnamurthy, and Arun Venkataramani. iplane: An information plane for distributed services. In *Proceedings of the 7th symposium on Operating Systems Design and Implementation (OSDI'06)*, Nov 2006.

[59] Harsha V. Madhyastha, Arun Venkataramani, Arvind Krishnamurthy, and Thomas Anderson. Oasis: An Overlay-Aware Network Stack. In *Operating Systems Review*, pages 41–48, 2006.

[60] Yun Mao, Bjorn Knutsson, Honghui Lu, and Jonathan M. Smith. DHARMA: Distributed Home Agent for Robust Mobile Access. In *Proceedings of Annual Joint Conference of the IEEE Computer Communications Societies (INFOCOM)*, 2005.

[61] Z. Morley Mao and Randy H. Katz. Achieving service portability using self-adaptive data paths. In *IEEE Communications Magazine special Issue on Service Portability and Virtual Home Environment*, Jan 2002.

[62] N. Milanovic and M. Malek. Current solutions for web service composition. *IEEE Internet Computing*, 8(6):51–59, Nov 2004.

[63] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[64] Jonathan T. Moore, Michael Hicks, and Scott Nettles. Practical programmable packets. In *Proceedings of Annual Joint Conference of the IEEE Computer Communications Societies (INFOCOM)*, pages 41–50, April 2001.

[65] Akihiro Nakao, Larry Peterson, and Andy Bavier. A Routing Underlay for Overlay Networks. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, 2003.

[66] Ryan Newton and Matt Welsh. Region streams: Functional macroprogramming for sensor networks. In *Proceedings of the First International Workshop on Data Management for Sensor Networks (DMSN)*, August 2004.

[67] T. S. Eugene Ng, Ion Stoica, and Hui Zhang. A waypoint service approach to connect heterogeneous internet address spaces. In *Proceedings of the USENIX Annual Technical Conference (USENIX'01)*, Boston, MA, June 2001.

[68] Charles E. Perkins. *IP mobility support for IPv4*. RFC 3220, Internet Engineering Task Force, 2002.

[69] Larry Peterson, Scott Shenker, and Jon Turner. Overcoming the Internet Impasse Through Virtualization. In *Proceedings of ACM HotNets-III*, 2004.

[70] PlanetLab. Global testbed. http://www.planet-lab.org/.

[71] B. Quinn and K. Almeroth. IP Multicast Applications: Challenges and Solutions. RFC3170, 2001.

[72] Anand Rajaraman, Yehoshua Sagiv, and Jeffrey D. Ullman. Answering queries using templates with binding patterns. In *ACM Symposium on Principles of Database Systems (PODS)*, 1995.

[73] Raghu Ramakrishnan and Jeffrey D. Ullman. A Survey of Research on Deductive Database Systems. *Journal of Logic Programming*, 23(2):125–149, 1993.

[74] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[75] Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. Opendht: a public dht service and its uses. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, 2005.

[76] Dennis Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, October 1984.

[77] A. Rodriguez, C. Killian, S. Bhat, D. Kostic, and A. Vahdat. MACEDON: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI'04)*, March 2004.

[78] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[79] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end Arguments in System Design. In *Proceedings of the 2nd IEEE International Conference on Distributed Computing Systems*, pages 509–512, April 1981.

[80] Stefan Savage, Tom Anderson, Amit Aggarwal, David Becker, Neal Cardwell, Andy Collins, Eric Hoffman, John Snell, Amin Vahdat, Geoff Voelker, and John Zahorjan. Detour: A Case for Informed Internet Routing and Transport. In *IEEE Micro*, Jan 1999.

[81] E. Shi, D. Andersen, A. Perrig, and I. Stoica. OverDoSe: A Generic DDoS solution using an Overlay Network. Technical Report CMU-CS-06-114, Carnegie Mellon University, 2006.

[82] Skype. Skype P2P Telephony. 2006. http://www.skype.com.

[83] Jonathan M. Smith and Scott M. Nettles. Active Networking: One View of the Past, Present and Future. *IEEE Transactions on Systems, Man, and Cybernetics*, 34(1):4–18, Feb 2004.

[84] Ion Stoica, Daniel Adkins, Shelley Zhuang, Scott Shenker, and Sonesh Surana. Internet Indirection Infrastructure. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, 2002.

[85] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, 2001.

[86] Lakshminarayanan Subramanian, Ion Stoica, Hari Balakrishnan, and Randy Katz. OverQoS: An Overlay Based Architecture for Enhancing Internet QoS. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI'04)*, 2004.

[87] David L. Tennenhouse, W. David Sincoskie Jonathan M. Smith, David J. Wetherall, and Gary J. Minden. A Survey of Active Network Research. *IEEE Communications*, 35(1):80–86, January 1997.

[88] J. Touch. Dynamic Internet overlay deployment and management using the X-Bone. In *Proceedings of the Eighth Annual International Conference on Network Protocols (ICNP)*, pages 59–68, Osaka, Japan, 2000.

[89] Jonathan S. Turner, Patrick Crowley, John DeHart, Amy Freestone, Brandon Heller, Fred Kuhns, Sailesh Kumar, John Lockwood, Jing Lu, Michael Wilson, Charles Wiseman, and David Zar. Supercharging PlanetLab: A High Performance, Multi-Application, Overlay Network Platform. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, pages 85–96, Kyoto, Japan, August 2007.

[90] Amin Vahdat and David Becker. Epidemic routing for partially-connected ad hoc networks. *Duke Technical Report CS-2000-06*, 2000.

[91] Limin Wang, Vivek Pai, and Larry Peterson. The Effectiveness of Request Redirecion on CDN Robustness. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*, Boston, MA USA, December 2002.

[92] David J. Wetherall, John Guttag, and David L. Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In *IEEE OpenArch Proceedings*. IEEE Computer Society Press, Los Alamitos, April 1998.

[93] J. T. Wroclawski. The Metanet. In *Proc. Workshop on Research Directions for the Next Generation Internet*, 1997.

[94] James Yonan. OpenVPN: Building and Integrating Virtual Private Networks. http://www.openvpn.net.

[95] Ming Zhang, Junwen Lai, Arvind Krishnamurthy, Larry Peterson, and Randolph Wang. A transport layer approach for improving end-to-end performance and robustness using redundant paths. In *Proceedings of the USENIX Annual Technical Conference (USENIX'04)*, 2004.

[96] Ben Y. Zhao, Ling Huang, Anthony D. Joseph, and John D. Kubiatowicz. Rapid mobility via type indirection. In *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS'04)*, San Diego, CA, February 2004.

[97] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report CSD-01-1141, U. C. Berkeley, Apr 2001.

[98] Li Zhuang, Feng Zhou, Ben Y. Zhao, and Antony Rowstron. Cashmere: Resilient anonymous routing. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI'05)*, April 2005.

[99] Shelley Q. Zhuang, Kevin Lai, Ion Stoica, Randy H. Katz, and Scott Shenker. Host Mobility using an Internet Indirection Infrastructure. In *Proceedings of ACM/Usenix Mobisys*, 2003.